ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

PROYECTO FIN DE GRADO

TÍTULO: Correlación de medidas de diversos sistemas de sensores inalámbricos a través del uso de Smartphone

AUTOR: Daniel Escribano García

TITULACIÓN: Grado en Ingeniería Telemática

TUTOR (o Director en su caso): Ana Belén García Hernando

DEPARTAMENTO: DIATEL

VºBº

Miembros del Tribunal Calificador:

PRESIDENTE: Juan Manuel López Navarro

VOCAL: Ana Belén García Hernando

SECRETARIO: Mary Luz Mouronte López

Fecha de lectura:

Calificación:

El Secretario,

Agradecimientos

Quiero agradecer a todos aquellos que me han ayudado a llegar hasta el día de hoy.

En primer lugar, quiero agradecer a mis padres, los cuales han dedicado todo su esfuerzo en conseguir lo mejor para mí, me han ayudado a tomar las decisiones correctas en cada momento difícil y me han dado un cariño indescriptible que solo ellos son capaces de dar.

A mi hermana, María, con la que he tenido grandes momentos.

A mi novia, Cris, que me ha apoyado en todo momento durante toda la carrera.

A mi amigo Alber, el cual ha sido mi compañero durante todo este viaje y me ha ayudado en todo lo que ha estado a su alcance. A mi amigo Valero, con el que he tenido y sigo teniendo grandes momentos de risas. A mi amigo Juanjo, que ha sido un hermano en cada uno de mis 23 años.

A mi tutora, Ana Belén, por haber hecho todo lo que ha tenido en su mano para permitirme presentar este proyecto y ayudarme a hacerlo lo mejor posible.

Gracias a todos.

Resumen

Este Proyecto Fin de Grado se centra en la definición de unos interfaces y el desarrollo de unos módulos que los ofrezcan y que permitan desarrollar un sistema para Smartphone mediante el que se puedan obtener medidas tanto de dispositivos biométricos como de una red inalámbrica de sensores (WSN – Wireless Sensor Network). Estos dos tipos de medidas deben poder ser mostradas de manera que se observe gráficamente su correlación espacio-temporal.

Por tanto, estos interfaces ofrecen, principalmente, la posibilidad de gestionar un número indeterminado de dispositivos biométricos y tomar medidas de ellos, además de mecanismos de almacenamiento para dichas medidas. También existe la posibilidad de crear una representación gráfica de dichas medidas. Por último, se desarrolla un interfaz para obtener información proveniente de una red de sensores inalámbricos instalada en un determinado entorno en el cual el usuario estará realizando sus propias medidas.

Además, se lleva a cabo la creación de la aplicación comentada, que hace uso de las interfaces especificadas, para realizar la correlación de las medidas. La aplicación permite al usuario mantener una lista de dispositivos, pudiendo consultar los parámetros de configuración de los mismos y tomar las medidas de aquellos que desee. Podrá visualizar en todo momento las medidas que se van realizando, y, por último, podrá representarlas gráficamente en pantalla.

Los interfaces están creados de forma que sean flexibles de modo que puedan añadirse nuevas funciones en un futuro y permitan ser utilizados para diferentes aplicaciones. Los módulos que ofrecen estos interfaces están desarrollados para cumplir todas las funcionalidades que esperamos llevar a cabo en la aplicación creada.

Abstract

This Final Degree Project is focused on the definition of a set of interfaces, together with the implementation of the modules that comply with them, with the aim of creating a smartphone-based system to obtain measurements from both biometric devices and a wireless sensor network (WSN). These two types of measurements have to be graphically shown in order to observe their spatial and temporal correlation.

Thus, the main purpose of the aforementioned interfaces is to manage an indeterminate number of biometric devices in order to obtain and store the measurements provided by them. There is also the possibility of creating a graphical representation of the data. In addition to all this, an interface has been developed for obtaining the information coming from a wireless sensor network deployed in the area where the user is taking his/her measurements.

Also as part of the work performed, the smartphone application that utilizes the specified interfaces has been implemented, in order to actually perform the measurements correlation. This application allows the user to maintain the biometric devices list and control their configuration, including the activation of the measurements taking process. These data can be visualized anytime and, moreover, they can be represented graphically in the smartphone screen.

The design of the interfaces is flexible in the sense that new functionality may be easily added to them in the future and new applications with different purposes may make use of them. The modules implemented as part of this Final Degree Project have been developed in order to comply with all the requirements of the smartphone system described above.

Contenido

Índice de Figuras

CAPÍTULO 1

INTRODUCCIÓN Y OBJETIVOS

1.1. Introducción

El proyecto “Correlación de medidas de diversos sistemas de sensores inalámbricos a través del uso de Smartphone”, nace de la intención de conectar sensores inalámbricos con los ya ampliamente extendidos Smartphones, de manera que podamos controlarlos desde éstos.

Cada día es más común la automatización y el seguimiento de todo tipo de información en tiempo real. En la actualidad los sensores inalámbricos son una realidad mediante la cual podemos recopilar gran cantidad de información de cualquier índole para posteriormente procesarla y adaptarla al uso deseado. Las redes de sensores inalámbricos se pueden aplicar a cualquier entorno imaginable: militar, naturaleza, salud o el hogar son algunas de los más comunes, y dentro de cada uno de estos ámbitos las posibilidades son innumerables.

Si bien hay diversos estudios y proyectos que tratan sobre la creación de espacios con inteligencia ambiental mediante la medición de magnitudes físicas a través de nodos sensores y el procesamiento de esta información para obtener conclusiones o llevar a cabo acciones sobre el entorno, también existen otros análisis que se centran más en la medición de parámetros fisiológicos (a través de sensores que se pueden "llevar encima" o incluso a través de lo que a veces se llama ropa inteligente) de manera que se puedan controlar datos de interés para la práctica de un deporte, para el control de la salud o para el confort de la persona.

En este proyecto pretendemos realizar una prueba de concepto sobre un sistema que integre estos dos tipos de aplicaciones, de manera que sea posible almacenar mediciones tanto ambientales como fisiológicas con indicación de la localización e instante correspondiente. La Figura 1 muestra los principales elementos del sistema desarrollado, que serán convenientemente explicados en posteriores capítulos de este documento.

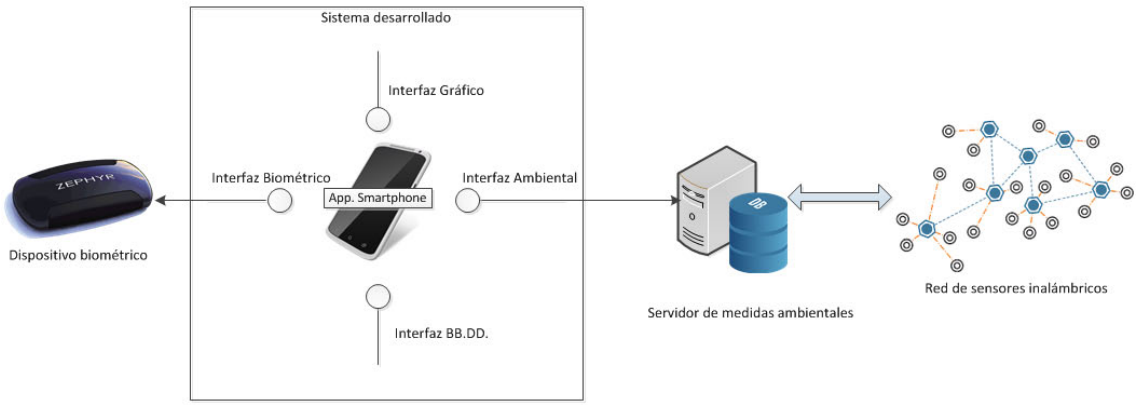

Figura 1: Sistema desarrollado en este proyecto

1.2. Objetivos

La idea principal de este Proyecto Fin de Grado es la definición de un conjunto de interfaces e implementación de módulos que las ofrezcan, desarrollando estos módulos de manera que facilitemos la creación de una aplicación para *smartphone* capaz de representar dos tipos de medidas en el mismo eje temporal y con indicación de la localización. El sistema operativo seleccionado para estos desarrollos es Android.

Por tanto, los objetivos principales de este PFG son los siguientes:

- Diseñar un interfaz capaz de manejar cualquier dispositivo biométrico, independientemente del funcionamiento concreto del mismo.
- Diseñar un interfaz capaz de almacenar medidas biométricas en una base de datos local, con indicación del instante y localización en los que fueron tomadas.
- Diseñar un interfaz que permita obtener información ambiental proveniente de una red de sensores inalámbricos.
- Diseñar un interfaz que permita visualizar gráficamente dos tipos de medidas, permitiendo además ver la correlación espacio-temporal entre las mismas.
- Permitir que cada interfaz pueda utilizarse con independencia del resto.
- Diseñar e implementar unos módulos que ofrezcan los interfaces mencionados.
- Diseñar e implementar una aplicación Android que permita observar los mecanismos de dichos interfaces, capaz de correlar medidas de una red de sensores inalámbricos y medidas de dispositivos biométricos.

1.3. Estructura de la memoria

El presente documento está estructurado en 6 capítulos y un anexo.

En este primer capítulo, “Introducción y objetivos”, se describen los objetivos que se persiguen con el proyecto. En el segundo capítulo, “Tecnologías relacionadas”, se recogen brevemente las características fundamentales de las tecnologías relacionadas con el trabajo desarrollado.

En el capítulo tres, “Descripción del sistema”, se explica la funcionalidad del sistema para posteriormente detallar cómo se han implementado los diferentes interfaces que lo componen, así como una explicación en profundidad de en qué consiste la aplicación y qué interfaces utilizan sus diferentes funcionalidades.

En el capítulo cuarto, “Pruebas”, se describen las pruebas realizadas a la aplicación desarrollada junto a los resultados obtenidos.

El capítulo quinto, “Conclusiones y trabajos futuros”, incluye las conclusiones obtenidas tras el desarrollo de este trabajo y algunas propuestas de futuras mejoras que pueden implementarse.

El capítulo sexto recoge la bibliografía utilizada.

Para finalizar, se recoge en un anexo la documentación *Javadoc* de los módulos desarrollados.

CAPÍTULO 2

TECNOLOGÍAS RELACIONADAS CON EL PROYECTO

En este capítulo se describen las bases tecnológicas y conceptuales que han servido como fundamento para la realización de este Proyecto Fin de Grado.

Hablaremos en primer lugar de las Redes de Sensores Inalámbricos. Consideramos de interés explicar esta tecnología debido a que la parte servidora de este proyecto consiste en una red de sensores ambientales distribuidos en un determinado entorno, los cuales realizan distintas mediciones del ambiente, como puede ser la temperatura o la humedad.

A continuación se hablará de los sistemas operativos móviles más relevantes hoy día, explicando brevemente la historia de cada uno de ellos, y realizando una comparativa describiendo sus ventajas e inconvenientes.

Posteriormente se hablará con más detalle de Android, por ser el sistema operativo escogido para la realización de este trabajo, describiendo sus características más importantes.

Para finalizar, se describen los dispositivos llevables y, en concreto, el modelo de dispositivo utilizado para realizar la aplicación de este proyecto.

2.1. Redes de Sensores Inalámbricos

Las redes de sensores inalámbricos están formadas por un grupo de sensores con capacidades sensitivas y de comunicación inalámbrica. Son capaces de medir magnitudes como temperatura, ruido o intensidad lumínica, entre otras.

Un sensor inalámbrico no sólo se encarga de realizar mediciones, sino que también se encarga de procesar, comunicar y almacenar. Con estas capacidades, un sensor, por lo general, no se encargará sólo de la recolección de datos, sino también de realizar análisis dentro de la red, correlacionar y comparar sus propios datos con los de otros nodos. Cuando múltiples sensores inalámbricos de estas características cooperan para monitorizar un determinado entorno, forman la denominada red de sensores inalámbricos (conocidas por sus siglas WSN en inglés, *Wireless Sensor Networks*). [**1**]

En la Figura 2 se puede observar el aspecto de un sensor inalámbrico de un fabricante concreto.

Figura 2: Nodo inalámbrico de V-Link [2]

Los sensores no sólo se comunican entre ellos, sino que también pueden hacerlo con una denominada estación base (o sumidero), la cual se encarga de enviar la información recogida por los diferentes sensores a un servidor para su posterior procesado.

El diseño de una determinada red de sensores se ve influido por los siguientes factores [**3**]:

Tolerancia a fallos

> Debido a que algunos nodos podrían fallar debido a falta de energía o algún daño físico, la red debe diseñarse de forma que, en caso de aparecer fallos en alguno de los nodos (o en varios de ellos), la red pueda seguir funcionando sin verse excesivamente afectada.

Escalabilidad

El número de nodos según qué aplicaciones, puede llegar a ser de miles o incluso de millones, por tanto la red debe ser capaz de adaptarse al crecimiento sin perder calidad y manejar el incremento de trabajo de manera fluida.

Coste de producción

Dado que la red se compone de un gran número de nodos, el coste de uno de ellos es importante a la hora de justificar el coste total. Es por ello que, si el coste de la red fuese más elevado que el de desplegar una sensores tradicionales, no se justificaría el construir una red de sensores inalámbricos.

Topología de red

Dado el alto número de nodos inaccesibles que pueden existir, realizar una topología que pueda mantenerse correctamente es otro factor importante en el diseño de la red. Una WSN está, en general, organizada en uno de los tres tipos de topología siguientes: topología de estrella, de árbol o de malla, ofreciendo cada una sus ventajas según el contexto de aplicación. No obstante, la topología de las WSN suele ser dinámica, pudiendo cambiar con el tiempo.

Entorno

Los nodos pueden desplegarse en una gran variedad de entornos, cerca o directamente dentro del fenómeno que quiere observarse. Pueden situarse en océanos, en lugares con huracanes o tornados e incluso en animales. Es por ello que este factor se tiene en cuenta a la hora de realizar el diseño.

Consumo

Por último, y no menos importante, otro factor determinante es el consumo energético. Los nodos, siendo dispositivos electrónicos, pueden estar equipados con una fuente de alimentación limitada. En determinados escenarios, se hace imposible reemplazar esta fuente cuando se agote. El consumo de los nodos se divide entre realizar las mediciones, comunicar, y procesar datos. Dependiendo del entorno en el que se vayan a desplegar, interesará más eficiencia en una determinada tarea que en otra, para optimizar el consumo, aunque normalmente es la comunicación lo que más energía consume.

Tras ver qué factores se tienen en cuenta para diseñar una red, veremos por último cómo es la arquitectura de comunicación de las redes de sensores, la cual puede verse en la Figura 3.

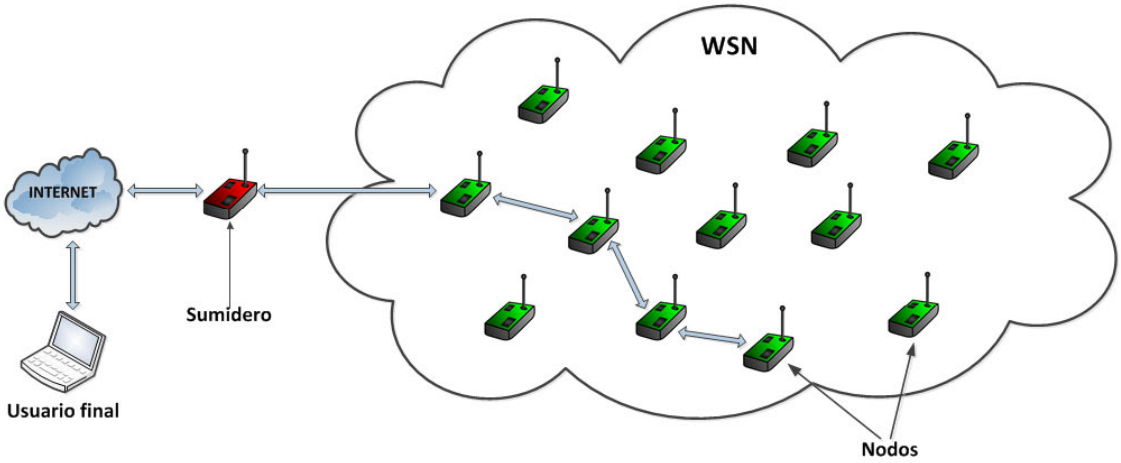

Figura 3: Ejemplo de Red de Sensores Inalámbricos

Como vemos, los nodos situados en el área donde se encuentra desplegada la red de sensores envían la información al sumidero enrutándola en general a través de otros nodos existentes, utilizando el camino más adecuado según el mecanismo de encaminamiento que se esté utilizando. El sumidero es el encargado de comunicarse con el usuario final a través, en general, de una red de área extensa (que puede ser por ejemplo Internet). Tanto el sumidero como el resto de nodos utilizan una pila de protocolos. En la Figura 4 se puede ver un ejemplo de pila de protocolos que puede aplicarse a WSN diseñadas para diversos propósitos.

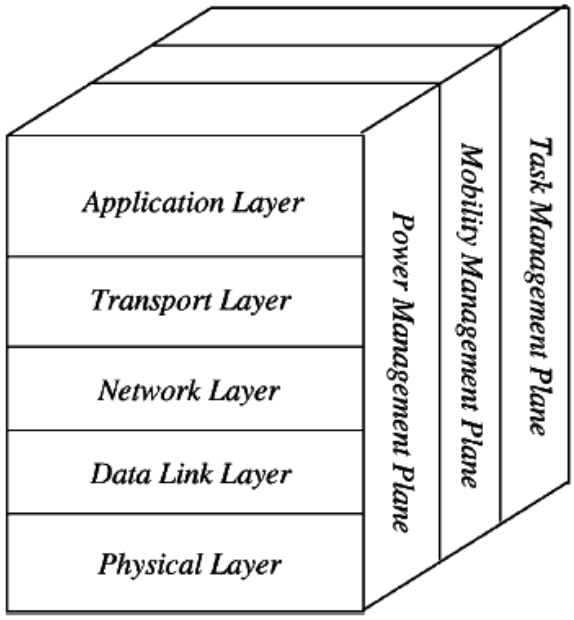

Figura 4: Ejemeplo de pila de protocolos de los sensores inalámbricos [3]

La pila de protocolos consiste en los niveles de aplicación, transporte, red, enlace y físico; junto a los planos de gestión energética, gestión de la movilidad y gestión de tareas.

Cada capa tiene sus funciones perfectamente diferenciadas. Por ejemplo, en la capa de aplicación se pueden construir diferentes aplicaciones. La de transporte, en caso de existir, ayuda a mantener un flujo de datos. La de red se encarga del encaminamiento, la de enlace del control de acceso al medio y la física de transmitir y recibir los datos a y del medio físico.

Por último, los planos de energía, movilidad y tareas monitorizan la energía, el movimiento (para el caso de una WSN en la que los nodos pueden desplazarse) y la distribución de tareas entre los distintos nodos. Estos planos ayudan a los sensores a coordinar las tareas de medición de manera cooperativa y a minimizar el consumo energético de la totalidad de la red. [**4**]

2.2. Sistemas operativos móviles

Dado que este trabajo está enfocado en software para un Smartphone, se antoja necesario un estudio previo de los diferentes sistemas operativos móviles que existen en el mercado actual, estudiando con detenimiento las ventajas e inconvenientes que ofrece cada uno de ellos.

2.2.1. Breve historia

Los sistemas operativos móviles (de ahora en adelante, S.O.) tal como los conocemos hoy día son una tecnología muy reciente.

El primer S.O. dominante fue Symbian. Symbian es un sistema operativo propiedad de Nokia, y que en el pasado fue producto de la alianza de varias empresas de telefonía móvil, entre las que se encontraban Nokia, Sony Mobile Communications, Samsung, Siemens, LG o Motorola entre otras.

Fue diseñado con el objetivo de competir con el S.O. de Palm o Windows Mobile en la época, y posteriormente con los actuales Android e iOS. No obstante, en octubre de 2011 se confirma de forma oficial que Symbian tendrá soporte hasta el año 2016, al no poder seguir soportándolo por no ser un competidor para la nueva versión de Smartphones con sistemas operativos de última generación como Android, iOS o Windows Phone.

El verdadero punto de inflexión respecto a los S.O. así como los Smartphone llega en 2007 con la aparición del iPhone y su S.O. iOS [**5**]. Apple sentó las bases de lo que serían los Smartphones tal como los conocemos hoy día: pantalla táctil, ausencia de teclado físico y acelerómetro para actuar según la posición del terminal.

Igualmente, en el año 2007, fue presentado Android [**6**]. Fue desarrollado inicialmente por Android Inc., una firma comprada por Google en 2005. Es el principal producto de la Open Handset Alliance [**7**], un conglomerado de fabricantes y desarrolladores de hardware, software y operadores de servicio.

Por último, entre los sistemas móviles actuales cabe destacar el de Microsoft, Windows Phone [**8**]. Microsoft mostró Windows Phone por primera vez el 15 de febrero de 2010. La versión final de Windows Phone 7 se lanzó el 21 de octubre de 2010 en Europa y el 8 de noviembre en Estados Unidos. Inicialmente, Windows Phone estaba destinado para ser lanzado durante el 2009, pero varios retrasos provocaron que Microsoft desarrollara Windows Mobile 6.5 como una versión de transición.

2.2.2. Comparativa de iOS, Android y Windows Phone

Haremos aquí una comparativa respecto a desarrollo en las tres principales plataformas que tenemos a nuestra disposición.

En primer lugar, comenzaremos con iOS. En iOS se hace necesario tener un Mac con la última versión de Mac OS disponible para poder desarrollar una aplicación, por lo que en términos de accesibilidad el coste se hace muy elevado para alguien que no disponga previamente de un equipo de esta marca. Sin embargo, el sistema operativo de Apple tiene una mejor gestión de memoria junto a un IDE (Entorno de Desarrollo Integrado - *Integrated Development Environment*) con un consumo de memoria bajo y un simulador muy fluido para poder realizar las pruebas de la aplicación desarrollada. El lenguaje para el desarrollo de las aplicaciones es Objetive-C, que es un lenguaje orientado a objetos, creado como un superconjunto de C.

A continuación pasamos a describir las características de Android. Contrariamente a iOS, para el desarrollo en Android, se utiliza Eclipse, que es el IDE oficial soportado por la plataforma, junto al plugin ADT (Android Developer Tools). Eclipse es gratuito y está disponible tanto para Windows, como Linux y Mac, pudiendo instalar el SDK (Software Development Kit) en cualquiera de estas plataformas, y haciendo mucho más accesible el desarrollo en Android. El lenguaje utilizado para crear las aplicaciones es Java. Los inconvenientes que se presentan son que el IDE tiene un consumo elevado de recursos y el emulador que incluye es muy limitado y lento, por lo que en general es necesario disponer de un terminal móvil real con el sistema operativo instalado para probar convenientemente los desarrollos. Además, la gran diversidad de dispositivos sobre los que corre Android actualmente, hace que deba tratar de optimizarse la aplicación para asegurarse de que funcione de una manera fluida en la mayoría de dispositivos posible.

Por último, tenemos la plataforma Windows Phone, la cual destaca por su total fluidez tanto para terminales de gama alta como terminales de gama media o baja. Para desarrollar en esta plataforma se necesita Visual Studio, el cual existe en la denominada versión Express, que está disponible de forma gratuita. El lenguaje de desarrollo utilizado es C#, creado también por Microsoft, el cual presenta muchas similitudes con Java.

En la Tabla 1 se puede ver la comparativa de una manera más resumida.

Tabla 1: Comparativa de las tres plataformas [**9**]

	Android	**iOS**	**Windows Phone**
Herramientas de desarrollo	Lentas	Rápidas	Rápidas
Lenguaje	Java	Objective-C	C#, Visual Basic, C++
Documentación	Buena	Completa, bien estructurada	Dispersa y confusa
Aprobación de App	Ninguna	Tarda hasta 2 semanas	Tarda una semana
Plataformas soportadas para desarrollo	Linux, Windows, Mac	Mac	Windows

La aprobación de la App es el tiempo que se tarda desde que se sube la aplicación hasta que la comprueban y la aprueban para publicarla en la tienda de aplicaciones específica de cada plataforma.

Vista la comparativa, para nuestro trabajo nos hemos decantado por Android debido a, principalmente, tres factores: Su accesibilidad, su lenguaje de desarrollo y la gran comunidad existente. El desarrollo para Android es completamente gratuito; se utiliza Java, que es el lenguaje que mejor dominamos para la realización del proyecto; y, finalmente, gracias a la gran comunidad existente, cualquier tipo de duda que surja puede ser en principio más rápidamente resuelta.

2.3. Android

Android es un sistema operativo basado en el kernel de Linux diseñado principalmente para dispositivos móviles con pantalla táctil, como teléfonos inteligentes o tabletas. Tiene una cuota de mercado a nivel mundial de más del 50%, más del doble de su principal competidor, iOS. No obstante, hay países donde su cuota de mercado es mucho mayor, como en España, donde llega casi a un 90% como se observa en la Figura 5.

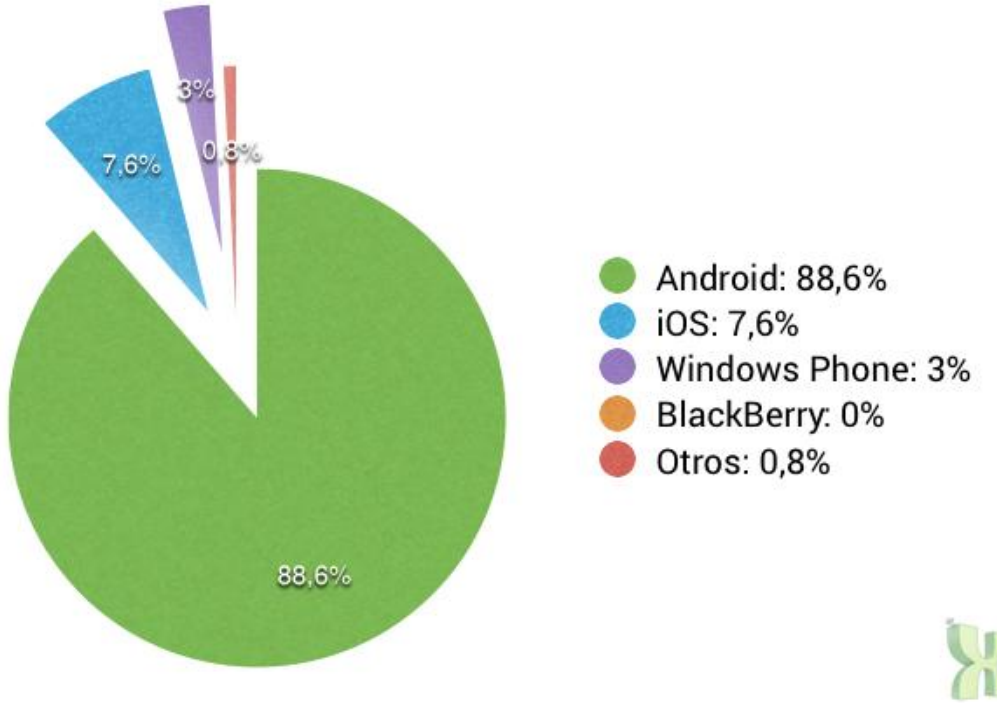

Figura 5: Cuota de mercado en España para las principales plataformas de Smartphone, en Marzo de 2014 [10]

Actualmente, gracias a la gran comunidad de desarrolladores que realizan aplicaciones para este sistema operativo, Android tiene a su disposición más de 1 millón de aplicaciones, de las cuales dos tercios son gratuitas.

Como se comentaba al comienzo de este apartado, Android está basado en el kernel de Linux, y su arquitectura está formada por varias capas que facilitan la creación de aplicaciones, abstrayendo al desarrollador de la programación de funcionalidades a bajo nivel. En la Figura 6 podemos observar este modelo de capas.

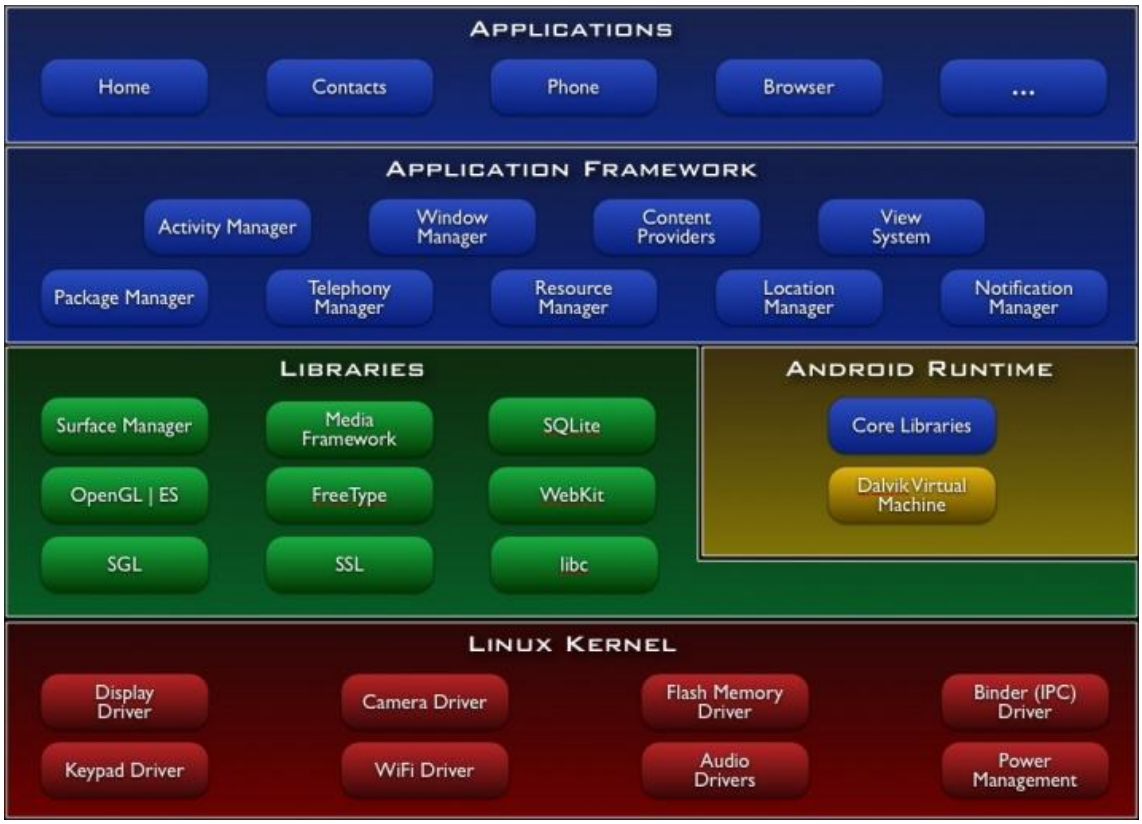

Figura 6: Arquitectura del sistema operativo Android [11]

En primer lugar, en la parte superior de la figura, tenemos la capa de aplicaciones (Applications). Aquí se incluyen todas las aplicaciones de dispositivo, tanto las preinstaladas como las que instala el usuario posteriormente. Una aplicación destacable en esta capa es la denominada Launcher (lanzador), que es la que permite ejecutar otras aplicaciones.

A continuación se encuentra la capa de framework de aplicaciones (Application Framework), en la cual se encuentran todas las clases y servicios de los que hacen uso las aplicaciones. Son, en general, librerías Java que acceden a recursos de capas inferiores.

Siguiendo el diagrama, nos encontramos con el entorno de ejecución (Android Runtime), que no es una capa en sí, sino que está formado por librerías con funciones típicas de Java. En el entorno de ejecución se encuentra la máquina virtual Dalvik. Dalvik es una variación de la máquina virtual de Java, pero no es compatible con el bytecode de Java, por lo que no se pueden ejecutar aplicaciones Java en Android ni viceversa.

Posteriormente está la capa de librerías (Libraries), las cuales están escritas en C o C++ y compiladas según el hardware específico del teléfono. Son realizadas por el fabricante de éste. Estas librerías incluyen OpenGL, bibliotecas multimedia, SSL o SQLite entre otras.

Por último, se encuentra el kernel de Linux (Linux Kernel). Éste actúa como una capa de abstracción entre el hardware y el resto de capas de la arquitectura. Como comentamos, el desarrollador accede a las librerías de capas superiores, por lo que éste no debe conocer características específicas de un determinado terminal para por desarrollar.

2.3.1. Evolución de Android

La primera versión de Android fue lanzada en Septiembre de 2008. Con el tiempo se han ido lanzando actualizaciones para corregir fallos de programa y agregar nuevas funcionalidades.

Estas actualizaciones tienen como peculiaridad que tienen como nombre en clave el nombre de diferentes postres y se van lanzando en orden alfabético. La primera versión fue la 1.0 Apple Pie, y actualmente, a fecha de realización de este proyecto, se encuentran en la 4.4 KitKat.

En la Figura 7, se puede observar la evolución del aspecto de la interfaz gráfica de usuario de Android desde la versión 1.0 hasta la actual 4.4:

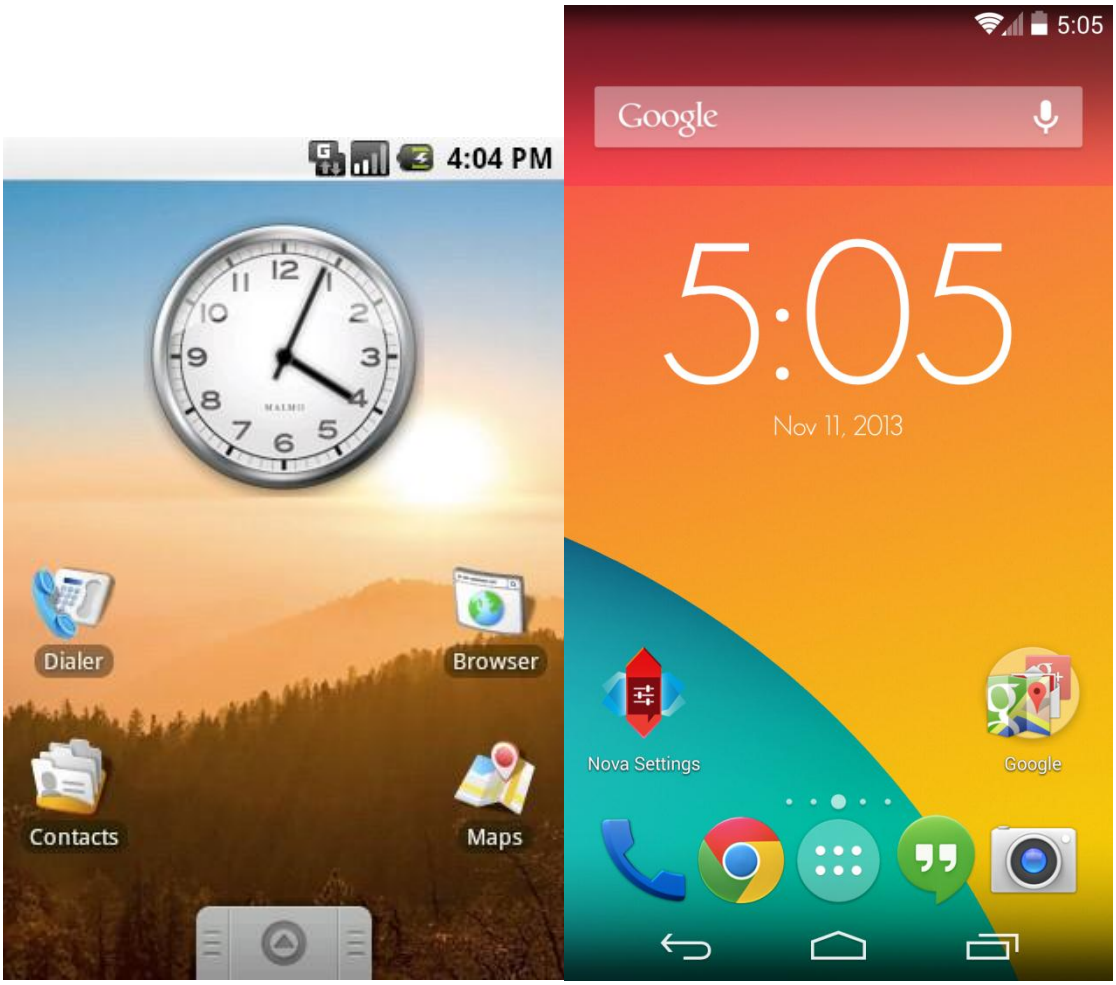

Figura 7: Diferencia de interfaz entre Android 1.0 (a la izquierda [12])y Android 4.4 (a la derecha [13])

2.3.2. Características principales

A continuación, en la Tabla 2, se expone la lista de características del sistema operativo que nosotros consideramos más relevantes para la realización del proyecto.

Tabla 2: Características de Android (elaborada a partir de la información presente en [**14**])

Diseño de dispositivo	La plataforma es adaptable a pantallas de altas resoluciones.
Almacenamiento	Utiliza SQLite como tecnología para el almacenamiento de datos.
Conectividad	Soporta tanto múltiples tecnologías WAN (como GSM/EDGE) así como tecnologías inalámbricas de área local (como WiFi).
Soporte de Java	Mediante la máquina virtual Dalvik, una variante de la máquina virtual Java, se compila el código Java en ésta para posteriormente ejecutarlo.
Soporte para hardware adicional	Android soporta cámaras de fotos, de vídeo, pantallas táctiles, o GPS, además de múltiples tipos de sensores.
Entorno de desarrollo	Incluye un emulador de dispositivos, herramientas para depuración de memoria y análisis del rendimiento del software. El entorno de desarrollo integrado es Eclipse usando el plugin de ADT.
Multi-táctil	Android tiene soporte nativo para pantallas capacitivas con soporte multi-táctil.
Bluetooth	Android soporta actualmente la versión de Bluetooth 4.0
Multitarea	Multitarea real de aplicaciones está disponible, es decir, las aplicaciones que no estén ejecutándose en primer plano reciben ciclos de reloj.

2.3.3. Desarrollo en Android

A continuación se describen las características principales del desarrollo específico de software para Android. Estas características deben ser conocidas por el programador, y han sido convenientemente utilizadas en el diseño e implementación de los distintos módulos generados durante la realización de este Proyecto Fin de Grado.

Las aplicaciones Android están escritas en Java. El SDK que proporciona Android se encarga de compilar todo el código junto a los ficheros de datos existentes, dando lugar a un fichero con extensión ".apk".

Una aplicación está definida por múltiples componentes. Los componentes de aplicación son bloques esenciales de la aplicación, cada uno es un punto diferente a través del cual el sistema accede a la aplicación y cada uno tiene su único rol en la aplicación. Existen cuatro tipos de componentes, y cada uno tiene un ciclo de vida que define cuándo es creado y cuándo destruido:

Actividades

Una actividad representa una pantalla con una interfaz de usuario. Múltiples actividades pueden formar una aplicación. Por ejemplo, en una aplicación de correo electrónico, una actividad podría usarse para mostrar una lista de correos mientras otra puede usarse para redactar un correo. Cada actividad es independiente de las otras y desde una se pueden lanzar las otras.

La interfaz gráfica de usuario se define a través de los denominados Layouts. Un layout es un fichero XML en el que se declaran los diferentes elementos de la interfaz.

Servicio

Un servicio es un componente que se ejecuta en segundo plano para realizar operaciones durante un largo periodo de tiempo. Un servicio no proporciona una interfaz de usuario. Un ejemplo sería un servicio que reproduce música en segundo plano mientras el usuario se encuentra en una aplicación diferente. Cualquier otro componente puede lanzar un servicio e interactuar con este.

Proveedor de contenido

Un proveedor de contenido administra un conjunto de datos compartidos. Los datos pueden ser almacenados en base de datos o en la memoria de almacenamiento del dispositivo, por ejemplo. A través de los proveedores de contenido, otras aplicaciones pueden modificar estos datos. También pueden usarse los proveedores para escribir o leer datos que son privados, y, por tanto, ninguna otra aplicación puede acceder a ellos.

Receptores broadcast

> Un receptor broadcast es un componente que responde a anuncios broadcast provenientes de todo el sistema (no solo de la aplicación). Un ejemplo esto podría ser un broadcast originado debido a que la pantalla se ha apagado o que la batería se está agotando. No obstante, las aplicaciones también pueden lanzar broadcasts. Aunque los receptores broadcast no tienen interfaz de usuario, pueden generar notificaciones para alertar al usuario de un determinado evento.

Para poder iniciar cualquiera de los componentes existentes en una aplicación, el sistema debe conocer de antemano qué componentes la forman. Para ello existe el fichero AndroidManifest.xml, en el cual se declaran todos los componentes. Además de ello, en este fichero se identifican los permisos que requiere la aplicación (como el acceso a Internet) y se declara el mínimo nivel de API (Application Programming Interface) necesaria para hacer funcionar la aplicación.

2.4. Dispositivos biométricos llevables

La La tecnología llevable consiste en ropa o accesorios que incorporan ordenador y electrónica avanzada.

Los dispositivos llevables están considerados como una de las tecnologías emergentes más destacadas, y entre ellos podemos encontrar desde gafas (como las actuales Google Glass [**15**]) hasta pulseras, bandas de *fitness*, o relojes; aunque en un futuro se espera adaptar la tecnología al cuerpo humano e implantar esta tecnología incluso debajo de la piel. En la Figura 8 se muestra un ejemplo del aspecto de algunos de estos dispositivos.

Figura 8: Ejemplos de dispositivos llevables [16]

Aunque existe un gran número de aplicaciones que pueden hacer uso de estos dispositivos, una en concreto es la que actualmente destaca por encima del resto y la que a nosotros nos es más relevante para nuestro proyecto: la monitorización del cuerpo (es decir, la obtención de medidas biométricas).

Dentro de la monitorización corporal existen también numerosas aplicaciones, como pueden ser desde un seguimiento de la salud del individuo hasta aplicaciones de *fitness*. En general, todas ellas miden ciertas constantes con las cuales se trabajan a la hora de realizar la aplicación. Ejemplos de estos dispositivos llevables son:

Ropa inteligente

Se trata de prendas de vestir que incluyen un sistema de monitorización de señales vitales. En general, suele tratarse de camisetas. Estas camisetas son capaces de medir las pulsaciones, el ritmo de la respiración o la temperatura corporal.

Pulseras cuantificadoras

Consisten en pulseras que lleva el usuario durante las 24 horas del día y son capaces de medir la actividad diaria que realiza. Son capaces de medir los pasos dados por el usuario a lo largo del día, el consumo calórico realizado e incluso llegan a monitorizar el sueño. No obstante, como su nombre indica, al ser cuantificadoras, estas pulseras únicamente se encargan de recopilar información, por lo que se hace necesario sincronizarlas con un Smartphone u ordenador personal para procesar esos datos y mostrárselos al usuario. La Figura 9 muestra una foto con un ejemplo de este tipo de dispositivo.

Figura 9: Ejemplo de pulsera cuantificadora [17]

Pulsómetros o bandas para el pecho

Consisten en bandas las cuales llevan un sensor capaz de medir parámetros como el ritmo cardiaco, la velocidad y la distancia recorrida por el usuario. Generalmente estas bandas se suelen vender con un reloj con el cual se sincronizan para mostrar la actividad al usuario, aunque también se venden solas y permiten sincronizarse con aplicaciones desarrolladas para dispositivos móviles. Se puede observar un ejemplo en la Figura 10.

Figura 10: Pulsómetro de la empresa Polar [18]

Relojes inteligentes

Aunque en principio no están pensados para ser usados como monitores corporales, existen modelos específicos desarrollados con este objetivo en mente. Tienen funciones similares a los anteriormente descritos, como medición del ritmo cardiaco, consumo calórico o seguimiento del sueño.

Aunque existen otros dispositivos llevables para monitorizar constantes corporales, consideramos que estos grupos son los que actualmente están más expandidos, si bien la ropa inteligente todavía no tiene un uso cotidiano.

De los comentados, el que hemos utilizado para nuestro proyecto es un pulsómetro, en concreto un modelo de Zephyr. Este modelo permite monitorizar las pulsaciones del usuario, y además ofrece una API para desarrollar aplicaciones que hagan uso del pulsómetro para sistema operativo Android. Se describe en el siguiente apartado.

2.4.1. Zephyr HxM BT

El Zephyr HxM BT [**19**] es un pulsómetro capaz de medir el ritmo cardiaco, la velocidad, distancia y nivel de intensidad de ejercicio. Está formado por el sensor encargado de comunicar mediante Bluetooth esta información y una banda conductora que se ajusta al pecho de la persona y que contiene el sensor biométrico propiamente dicho. Contiene una batería recargable que dura hasta 26 horas por cada carga. Su aspecto puede verse en la Figura 11.

Figura 11: Modelo BT HxM de Zephyr [20]

Una de las características más interesantes de este dispositivo es la posibilidad de desarrollar aplicaciones propias gracias a la API que ofrece. Mediante el uso de esta API se puede acceder a las medidas que realiza el dispositivo y procesarlas como nosotros deseemos según la aplicación que estemos desarrollando, por lo que no necesariamente debe utilizarse para *fitness*.

Esta API permite crear aplicaciones tanto en Android como en Windows Phone 8. Algunos ejemplos de las aplicaciones disponibles pueden verse en la Figura 12.

Figura 12: Aplicaciones que hacen uso del HxM [21]

CAPÍTULO 3

DESCRIPCIÓN DEL SISTEMA

3.1. Requisitos y casos de uso

3.1.1. Requisitos funcionales

Los requisitos funcionales son aquéllos que definen qué características debe poseer un determinado sistema para satisfacer unas determinadas necesidades. Es decir, son los que dan respuesta a la pregunta: ¿Qué necesidades se busca cumplir?

- Agregar, modificar o eliminar dispositivos biométricos propios de forma sencilla.

- Obtener medidas de todos los dispositivos agregados de manera convergente, es decir, a menos que sea de forma voluntaria, no tener la necesidad de crear diferentes funciones para recibir las medidas de los distintos dispositivos agregados.

- Gestionar una base de datos de medidas fácilmente, sin tener que poseer conocimientos de lenguajes para la gestión de bases de datos.

- Obtener medidas ambientales realizadas por una red de sensores inalámbricos que se encuentren almacenadas en un servidor remoto, de manera que necesite aportar los mínimos datos posibles sobre ese servidor (por ejemplo, su dirección IP).

- Creación de una gráfica que permita representar una serie de medidas, tanto ambientales como biométricas, tomadas en un intervalo de tiempo determinado, de forma que pueda verse claramente en qué instantes de tiempo se han realizado y en qué lugar estaba el usuario cuando se realizaron.

Una vez definidas las necesidades que buscamos satisfacer, podemos definir los requisitos funcionales del sistema:

- Permitir agregar dispositivos, modificar parámetros de éstos una vez añadidos o eliminarlos cuando así sea requerido.

- Permitir buscar dispositivos existentes en el entorno del usuario, ofreciendo transparencia a la hora de realizar búsquedas según la tecnología de la que hagan uso estos dispositivos. Es decir, con sólo indicar qué tecnología tiene el dispositivo a añadir, será suficiente.

- Ofrecer una librería que encapsule una base de datos para aportar la máxima facilidad a la hora de tener que almacenar medidas, ofreciendo métodos para conectar con ésta, agregar medidas, obtenerlas o eliminarlas de forma rápida.

- Dar la capacidad de obtener un listado de medidas en un intervalo de tiempo determinado.

- Conectar con un servidor remoto para obtener un listado de medidas ambientales en un intervalo de tiempo determinado.

- Permitir generar gráficas aportando simplemente una serie de medidas y unos intervalos de tiempo.

Como se puede observar, los requisitos buscan satisfacer las necesidades que pueda requerir el usuario en cualquier momento. Más adelante se definirán los casos de uso para cada interfaz, profundizando en las funciones que cada una aporta para cumplir la lista de requisitos dados.

3.1.2. Requisitos no funcionales

Los requisitos no funcionales son aquéllos que imponen restricciones en el diseño o la implementación sin estar directamente relacionados con funcionalidades que percibe directamente el usuario.

Extensibilidad

El sistema debe ser extensible, es decir, debe permitir implementar nuevas funciones con el paso del tiempo sin afectar a lo ya desarrollado anteriormente, así como adaptarse a necesidades que puedan surgir con el desarrollo de nuevos dispositivos de medidas. Consideramos este requisito como imprescindible, ya que debemos poder permitir el uso de nuevas tecnologías que puedan surgir en un futuro, así como ampliar funciones que puedan requerirse al hacer uso de dispositivos biométricos.

Sencillez

El sistema debe ser simple y fácil de utilizar, aportando la mayor documentación posible en aquellos casos en los que puedan presentarse ambigüedades.

Rendimiento

El sistema debe ser capaz de rendir adecuadamente en cualquier Smartphone Android de gama media (o alta) disponible en el mercado, evitando un elevado consumo de recursos.

3.1.3. Casos de uso

En este apartado se desglosan los casos de uso de cada interfaz, para ver con claridad las posibilidades que cada una ofrece. Para ello haremos uso de diagramas UML (Unified Modified Language) [**22**], así como una breve descripción textual de las funcionalidades.

3.1.3.1. Interfaz Biométrica

Es el interfaz principal del sistema, ya que es la que permitirá la gestión de los dispositivos biométricos (ver Figura 13).

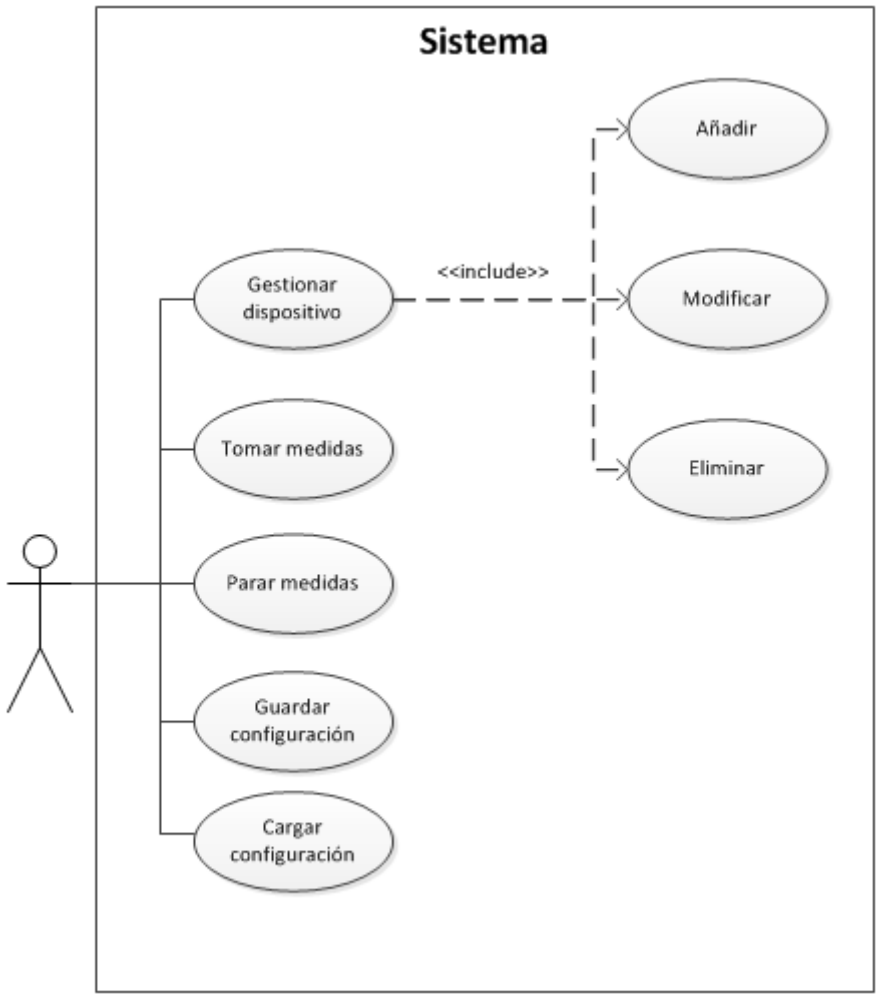

Figura 13: Casos de uso del Interfaz Biométrico

Añadir/modificar/eliminar dispositivos

Como el propio nombre indica, permitirá añadir dispositivos, así como modificar sus parámetros cuando sea necesario o eliminarlos cuando no se necesite usarlos más.

Tomar medidas

Iniciará la toma de medidas de un dispositivo biométrico concreto cada un determinado periodo de tiempo establecido por el usuario.

Parar medidas

Cancelará la toma de medidas del dispositivo indicado cuando así se requiera.

Guardar o cargar configuración

Se dará la posibilidad de guardar (o cargar) configuración de los dispositivos que haya añadido el usuario, para que no tenga que volver a introducirla cada vez que inicie la aplicación.

3.1.3.2. Interfaz Base de Datos

Es el interfaz encargado de encapsular una base de datos para ofrecer al usuario sencillez a la hora de almacenar medidas biométricas (ver Figura 14).

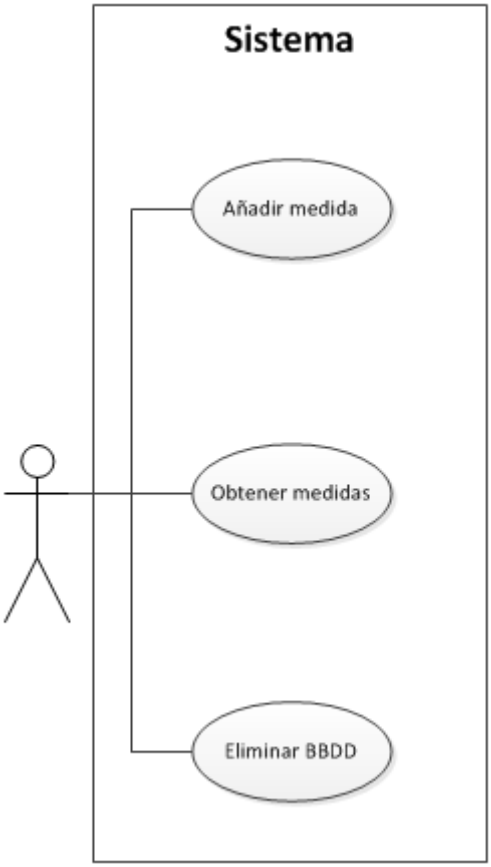

Figura 14: Casos de uso del Interfaz BB.DD.

Añadir medida

Añade una nueva medida a la base de datos.

Obtener medidas

Devuelve una lista de medidas. Puede obtenerse la lista completa o una lista según un intervalo de tiempo especificado por sus instantes inicial y final.

Eliminar BBDD

Elimina la base de datos del Smartphone.

3.1.3.3. Interfaz Ambiental

Se encargará de obtener una lista de medidas ambientales. Así, actuará de forma análoga al interfaz de base de datos, permitiendo obtener un listado de medidas, sólo que, en este caso, de tipo ambiental (ver Figura 15).

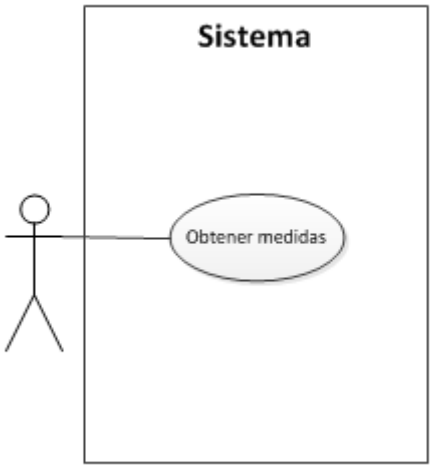

Figura 15: Casos de uso del Interfaz Ambiental

Obtener medidas

Obtendrá una lista de medidas ambientales en un intervalo de tiempo determinado.

3.1.3.4. Interfaz Gráfica

Proporciona métodos para mostrar simultáneamente hasta dos magnitudes de manera gráfica, compartiendo el mismo eje de tiempo y con indicación de la localización donde fueron obtenidas (ver Figura 16). Aunque está pensado para poder ofrecer una correlación entre dos tipos de medidas (biométricas y ambientales), el usuario podrá representar las medidas que desee.

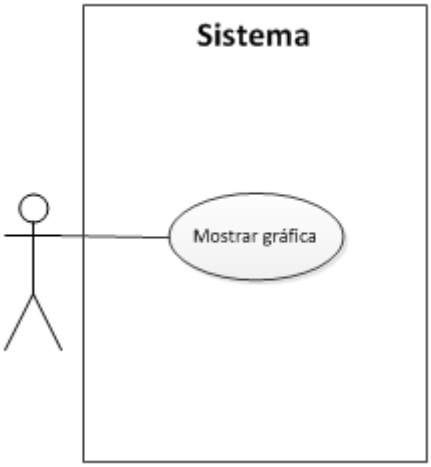

Figura 16: Casos de uso del Interfaz Gráfico

Mostrar gráficas

Dibujará una gráfica en pantalla con los datos aportados por el usuario.

3.2. Arquitectura del sistema

3.2.1. Diagrama de bloques

Los elementos que forman la arquitectura del sistema desarrollado se pueden dividir en tres grupos: Interfaces, módulos y aplicación móvil. Puede verse el diagrama de la arquitectura en la Figura 17.

Interfaces

Los interfaces ofrecerán las funciones generales que realizará el sistema. La persona encargada de implementarlas deberá adaptarlos para el uso concreto que desee darles, siempre y cuando respete las funciones ofrecidas por cada uno de ellos, descritas en el apartado 3.1.3., dedicado a los casos de uso.

Módulos

Los módulos hacen referencia a la implementación de los interfaces que hemos realizado nosotros para nuestro proyecto. Así, se respetan todas las funciones que estos ofrecen pero están adaptados para lograr los objetivos que perseguimos con la creación de la aplicación móvil.

Aplicación móvil

La aplicación móvil hace uso de los interfaces que ofrecen cada uno de los módulos desarrollados para lograr el objetivo descrito anteriormente, que es el de realizar la correlación de dos tipos de medidas sobre el mismo eje de tiempos.

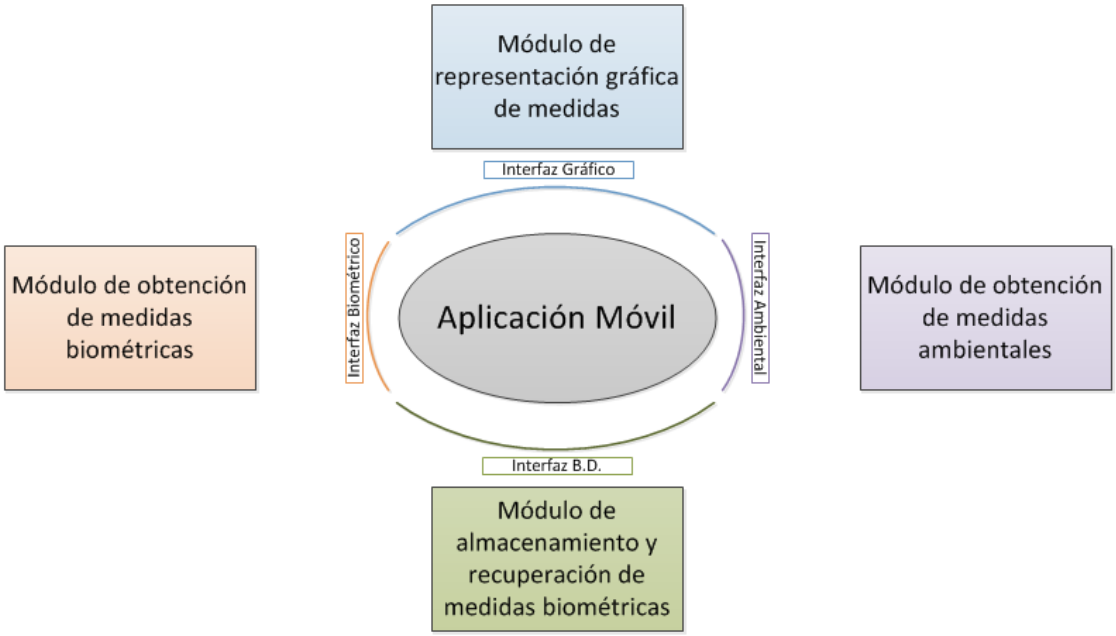

Figura 17: Arquitectura del sistema

3.2.2. Especificación de los interfaces

En este apartado se especifica cómo un programador puede hacer uso de los diferentes interfaces desarrollados. Si bien están desarrolladas en una misma librería, la independencia que tienen unos interfaces de otras permite al programador que vaya a hacer uso de la que necesite sin tener que utilizar el resto.

Se especificará con un alto nivel de detalle las posibilidades que ofrecen los interfaces, explicando cómo se han desarrollado, qué funcionalidad tienen y cómo se utilizan.

3.2.2.1. Especificación del Interfaz Biométrico

El interfaz biométrico, como se describió en el apartado 3.1.3.1., permite la gestión de dispositivos biométricos de cualquier tipo.

Este interfaz ofrece diferentes clases, siendo la principal la denominada Conector Biométrico, que ofrece todos los métodos necesarios para el manejo de los dispositivos.

El conector biométrico es capaz de gestionar dispositivos de medidas independientemente de su tecnología o funcionamiento a la hora de tomar medidas. Para ello, el interfaz tiene gestores internos encargados de manejar cada dispositivo adaptándose a su tecnología, aportando transparencia en este sentido. Además, al tener gestores según la tecnología indicada (ver Figura 18), se logra que el interfaz sea extensible, permitiendo que para el uso de nuevas tecnologías únicamente sea necesario desarrollar nuevos gestores. La Figura 18 muestra este concepto en un diagrama.

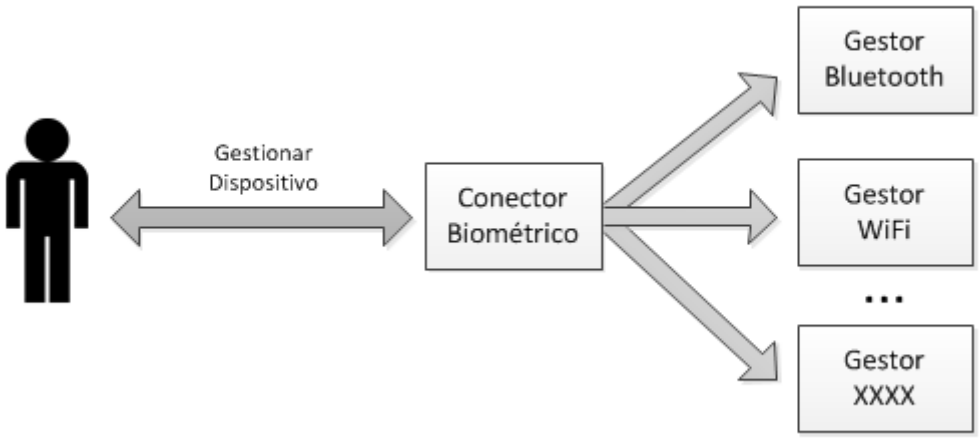

Figura 18: Gestión de tecnologías del interfaz biométrico

Para poder usar correctamente el conector biométrico a la hora de gestionar un dispositivo, este componente necesita conocer varios datos esenciales acerca del dispositivo para que todo funcione correctamente. Estos datos son: Su nombre, la tecnología que utiliza, el periodo con el que se desea obtener las medidas, un manejador de eventos y un driver.

El nombre será necesario para poder identificar al dispositivo. La tecnología permitirá al conector elegir qué gestor se va a encargar de realizar funciones propias de la misma. El periodo se utilizará para solicitar una medida al dispositivo cuando pase el tiempo determinado. El manejador de eventos servirá para comunicar cualquier evento relacionado con el dispositivo. Por último, el driver será utilizado para que el conector biométrico pueda comunicarse con el dispositivo.

Así, una vez tenemos el dispositivo identificado con sus diferentes parámetros configurados, sólo es necesario agregarlo al conector biométrico como se indica en el diagrama de secuencia de la Figura 19.

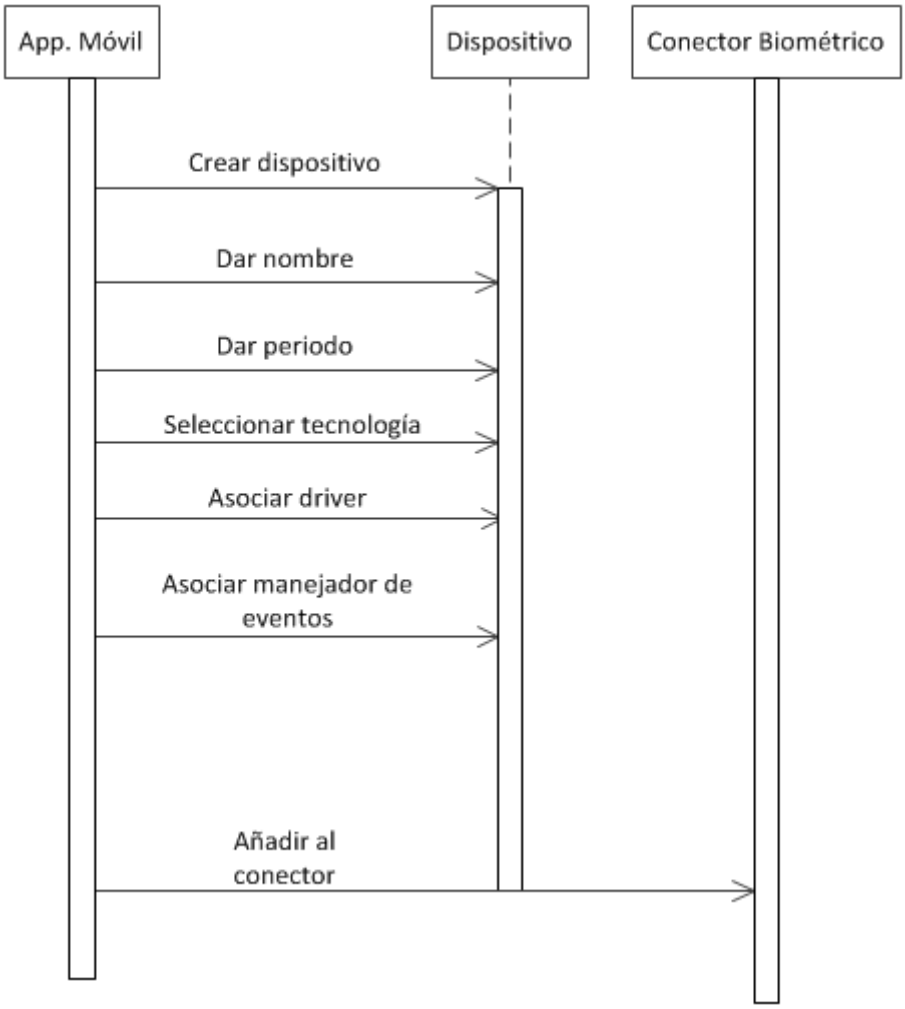

Figura 19: Adición de dispositivo al conector biométrico

Una vez añadido, se da por hecho que existe todo lo necesario para que, a través del conector biométrico, podamos iniciar o parar medidas del dispositivo, se nos comunique cualquier tipo de evento relacionado con éste y podamos modificar sus parámetros o incluso eliminarlo en un futuro.

No obstante, el conector no solo nos enviará eventos relacionados con el dispositivo. También puede que nos envíe un evento a nivel de sistema, como, por ejemplo, la necesidad de habilitar algún ajuste del teléfono para poder comunicarnos con el dispositivo. Por esto, a la hora de manejar eventos, se tienen en cuenta dos tipos:

- Eventos a nivel de sistema
- Eventos a nivel de dispositivo

Para poder procesar los primeros, el conector biométrico ofrece la opción de indicarle la referencia (función de *callback*) a través de la cual nos llegarán estos tipos de eventos.

Para los segundos, ya se ha indicado que, antes de agregar un dispositivo al conector, debe tener asociado un manejador de eventos propios del dispositivo.

Creemos que así que existirá más orden y menos confusión, comunicando los eventos particulares por un manejador concreto, de forma que el programador sea capaz de identificar qué dispositivo lo ha generado fácilmente. Asimismo, se consigue independencia, de forma que si un dispositivo falla, el resto no se verá afectado de ninguna manera. En la Figura 20 se puede ver un diagrama que ilustra con dos ejemplos el funcionamiento del sistema de comunicación de eventos.

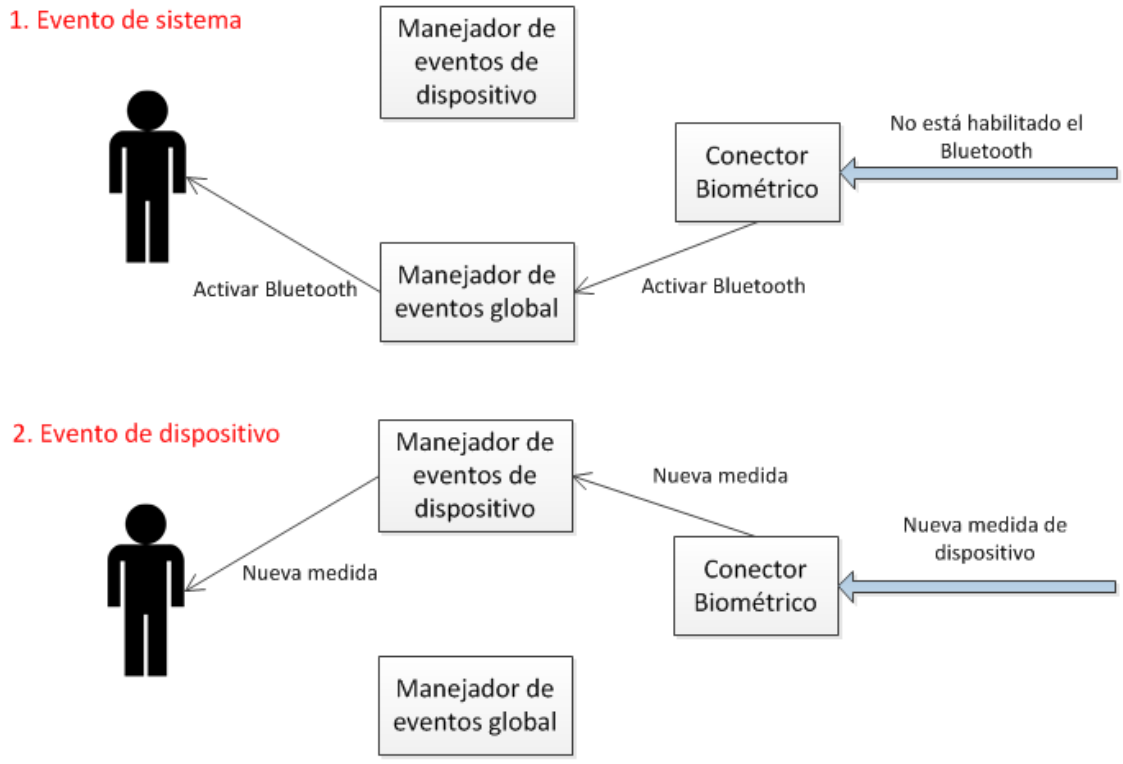

Figura 20: Tipos de eventos

Como vemos, el interfaz biométrico se encarga básicamente de tener un seguimiento de los diferentes dispositivos que añade el usuario, abstrayendo a éste de las funciones específicas de cada uno de ellos y tratándolos a todos como si fueran iguales, obviando sus diferencias. Simplemente, a la hora de configurarlo, será necesario asociar su driver y su manejador de eventos correspondiente, y el conector se encargará del resto.

3.2.2.2. Especificación del Interfaz Base de Datos

Este interfaz se ha desarrollado con el objetivo de dotar al programador de mecanismos para poder mantener una base de datos de medidas en el Smartphone desde el cual está gestionando los dispositivos biométricos, sin tener que precisar de conocimiento alguno en bases de datos. Es por ello que se puede considerar que este interfaz encapsula una base de datos.

Por tanto, el interfaz ofrece métodos directos para guardar medidas con indicación del instante y la localización en los que fueron tomadas, así como permitir recuperarlas cuando así sea preciso, ya sea todas ellas o en intervalos concretos. También permite eliminar la base de datos completamente (ver Figura 21).

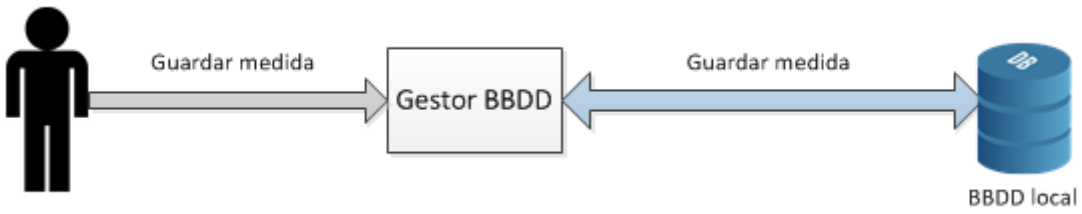

Figura 21: Uso del interfaz BB.DD.

3.2.2.3. Especificación del Interfaz Ambiental

El interfaz ambiental se centra principalmente en la petición de medidas realizadas por una serie de sensores inalámbricos distribuidos en un entorno. Por ejemplo, en una casa podrían existir sensores para realizar mediciones como temperatura, humedad o luminosidad.

Debido a que se quiere simplificar el proceso de obtención de las medidas, para tratar de conseguir transparencia en el uso, el programador simplemente tendrá que realizar la petición de las medidas de forma análoga a como se hace en el caso del interfaz de la base de datos, es decir, se obtendrán unas medidas según una localización y un intervalo de tiempo determinado. En el momento de realizar la implementación, deberá ocultarse cómo se realiza la petición de estas medidas, sin tener que involucrar a la persona que haga uso de la interfaz en este proceso, ya que de lo cotnrario no se conseguiría la simplicidad que deseamos.

La Figura 22 muestra de manera gráfica los componentes relacionados con el interfaz ambiental, utilizando como ejemplo de localización un identificador genérico que podría pertenecer a una casa (“Habitación”).

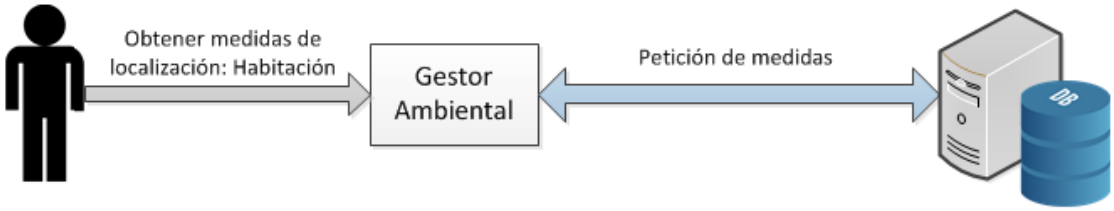

Figura 22: Uso del interfaz Ambiental

3.2.2.4. Especificación del Interfaz Gráfico

El interfaz gráfico tiene como función ofrecer a un programador la posibilidad de representar gráficamente las medidas tomadas por los dispositivos del usuario.

Estas medidas se representarán de forma que se pueda ver fácilmente en qué localizaciones se situaba el usuario así como los instantes en los que fueron tomadas.

Al igual que con los interfaces de base de datos y ambiental, el interfaz gráfico hace transparente al programador el proceso de construcción de la gráfica, teniendo únicamente que pedir la representación con las medidas que se le pasen, sin necesidad de profundizar más.

3.2.3. Diagramas de clases de los interfaces

En las siguientes páginas se muestran los diagramas de clases completos de cada uno de los interfaces. En ellos se puede ver los principales métodos y variables de cada interfaz.

3.2.3.1. Diagrama de clases del Interfaz Biométrico

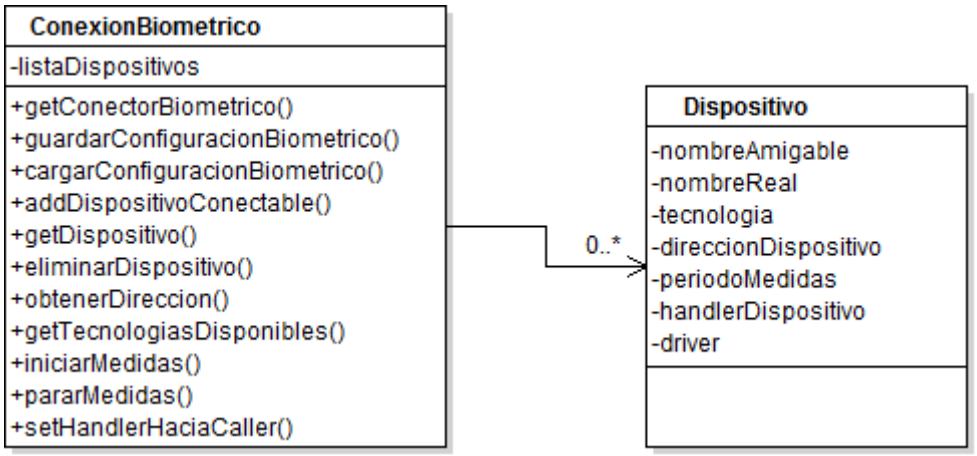

Figura 23: Diagrama de clases del Interfaz Biométrico

El interfaz biométrico ofrece los siguientes métodos para ser utilizados:

getConectorBiometrico

Permite obtener una instancia de conector biométrico.

guardarConfiguracionBiometrico

Guardará la configuración actual de todos los dispositivos agregados al conector biométrico.

cargarConfiguracionBiometrico

Cargará los dispositivos con la configuración indicada en un fichero de configuración guardado previamente.

addDispositivoConectable

Añade un dispositivo al conector.

getDispositivo

Obtiene un dispositivo.

eliminarDispositivo

Elimina un dispositivo del conector.

obtenerDireccion

Obtiene la dirección de un dispositivo biométrico.

getTecnologiasDisponibles

Obtiene una lista de tecnologías soportadas por el conector biométrico.

iniciarMedidas

Inicia la toma de medidas de un dispositivo.

pararMedidas

Detiene la toma de medidas de un dispositivo.

setHandlerHaciaCaller

Indica la referencia a través de la cual se enviarán los eventos de sistema.

A parte de los métodos, la interfaz también establece cómo debe ser el modelo de dispositivo biométrico con todos los parámetros básicos que necesita conocer para manejarlo correctamente.

3.2.3.2. Diagrama de clases del Interfaz Base de Datos

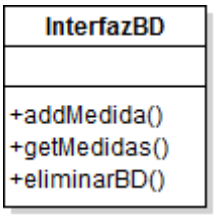

Figura 24: Diagrama de clases del Interfaz B.D.

El interfaz base de datos ofrece los siguientes métodos:

addMedida

Añade una nueva medida a la base de datos.

getMedidas

Obtiene una lista de medidas de la base de datos según un intervalo de tiempo determinado.

eliminarBBDD

Elimina la base de datos.

3.2.3.3. Diagrama de clases del Interfaz Ambiental

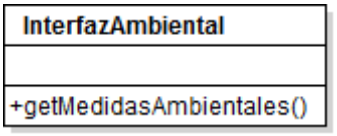

Figura 25: Diagrama de clases del Interfaz Ambiental

El interfaz ambiental ofrece un método únicamente:

getMedidasAmbientales

Permite obtener una lista de medidas ambientales entre un intervalo de tiempo, indicando su tipo y su localización.

3.2.3.4. Diagrama de clases del Interfaz Gráfico

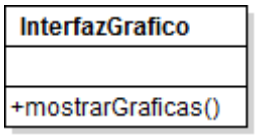

Figura 26: Diagrama de clases del Interfaz Gráfico

El interfaz gráfico ofrece únicamente un método:

mostrarGraficas

Representa una o dos gráficas que contienen una serie de medidas junto con los instantes en que se tomaron y su localización.

3.3. Implementación

3.3.1. Entorno de desarrollo

3.3.1.1. Eclipse

Eclipse [**23**] es el IDE que hemos elegido para desarrollar todo el código. Un IDE es un entorno de programación consistente en un editor de código, un depurador y un constructor de interfaz gráfica.

La razón de esta elección ha sido que el PFG está orientado a un sistema móvil para Android, y para el desarrollo en Android se necesita hacer uso de un plugin para eclipse denominado ADT (Android Development Tools).

En concreto, la versión de ADT utilizada es la 22.3.0

3.3.1.2. Android SDK

El SDK de Android está compuesto por un depurador de código, biblioteca y un emulador, así como ejemplos de código y tutoriales. La plataforma soportada de forma oficial es Eclipse junto con el plugin ADT.

Puede obtenerse de la página oficial [**24**], y con una descarga podemos obtener todas las herramientas necesarias para comenzar a desarrollar.

En concreto, la versión del SDK de Android utilizada es la 18.

3.3.1.3. Zephyr HxM BT

El Zephyr HxM BT [**19**] es un sensor biométrico orientado a la práctica deportiva, capaz de medir el ritmo cardíaco, velocidad, distancia recorrida y nivel de intensidad.

Ofrece una API para permitir el desarrollo de aplicaciones para Android y Windows Phone 8.

Se comunica con el dispositivo móvil a través de Bluetooth.

3.3.1.4. AChartEngine

AChartEngine [**25**] es una librería para realizar gráficos para Android.

Dado que no hay librerías oficiales para el soporte de gráficas, se ha tenido que recurrir a una de las múltiples opciones generadas por la comunidad de desarrolladores de Android. Hemos escogido AChartEngine dado a que es una de las que mayor soporte tiene en la comunidad y ofrece multitud de funciones a la hora de personalizar las gráficas.

El principal problema ha sido la falta de documentación oficial, por lo que se ha tenido que recurrir a foros, principalmente StackOverflow [**26**], donde el desarrollador de esta librería responde a todas las preguntas existentes.

3.3.2. Aplicación móvil desarrollada

La aplicación que hemos creado posibilitará a un usuario añadir tantos dispositivos biométricos como desee, siendo capaz de tomar medidas de todos ellos simultáneamente. La aplicación guardará las medidas tomadas por dichos dispositivos en una base de datos local, haciendo uso de la tecnología SQLite que ofrece Android. Para que el usuario pueda hacer un seguimiento de estas medidas, podrá ver un listado de las que se han guardado en la base de datos cuando así lo desee, así como eliminarlas en cualquier momento. Por último, la aplicación da la posibilidad de realizar una gráfica con las medidas biométricas tomadas y relacionarlas con medidas ambientales. Para ello, se tiene en cuenta la localización en la que se encuentra el usuario en el momento de tomar la medida, de forma que cuando se quiera representar la gráfica con medidas ambientales, se realiza la una petición para obtener la lista de medidas, indicando tanto la localización como el intervalo de tiempo de las medidas que se requieren.

La siguiente figura muestra el menú principal de la aplicación:

Figura 27: Menú de la aplicación

Podemos ver una aplicación sencilla, con un que consta de un menú de tres elementos:

- **Dispositivos**: Pantalla inicial de la aplicación. Es en ésta donde se pueden ver los dispositivos biométricos que tiene configurados el usuario y su estado, así

como poder editarlos o añadir nuevos. También puede, si así lo desea, guardar la configuración que tenga en ese momento para posteriormente cargarla, sin tener que volver a configurarlos manualmente.

- **Medidas**: Pantalla en la que se pueden visualizar las medidas que hay almacenadas en la base de datos local. Este listado muestra el tipo de medida y su valor, así como la fecha en la que fue tomada y la localización del usuario al momento de tomarse. También permite borrar todas las medidas.

- **Gráfica**: Tercer y último elemento del menú. Consta de dos secciones: una para la configuración de la gráfica y otra para mostrar la gráfica en sí.

Una vez ha sido descrito el menú principal de la aplicación, explicaremos qué uso se ha hecho de los interfaces para lograr construirla.

Pantalla Dispositivos

Se puede considerar a esta la pantalla la principal de la aplicación, ya que aquí es donde el usuario gestiona sus dispositivos biométricos. En la Figura 28 se puede observar su aspecto.

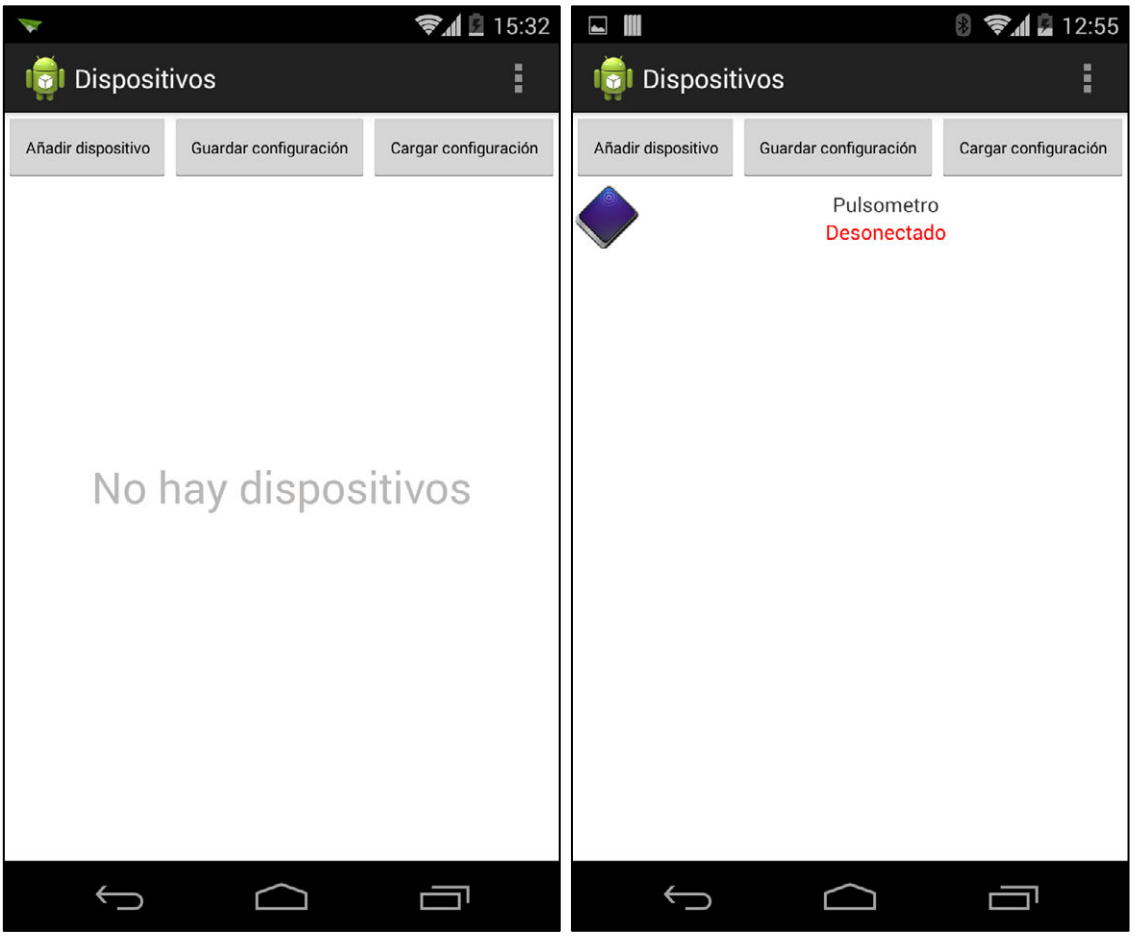

Figura 28: Pantalla de dispositivos vacía (izquierda) y con un dispositivo agregado (derecha)

La pantalla dispositivos hace uso del interfaz biométrico principalmente.

De entrada, cuando se carga la pantalla, se obtiene el denominado conector biométrico. Con este nombre hacemos referencia al objeto gestor de todo lo relacionado con los dispositivos biométricos, y el cual el programador usará a la hora de agregar dispositivos, eliminarlos, conectarse a ellos, y demás opciones propias de éstos.

Una vez tenemos la instancia del conector, obtenemos la lista de dispositivos biométricos existente y se la mostramos al usuario. Al ser cargada por primera vez en memoria, no existirán dispositivos. Sin embargo, si el usuario ya ha agregado dispositivos anteriormente, éstos se mostrarán como se aprecia en la imagen de la derecha de la Figura 28.

El usuario tiene ahora varias opciones: Agregar un nuevo dispositivo, editar uno existente (en caso de que haya), guardar configuración actual, cargar una configuración previamente guardada, o conectarse al dispositivo para iniciar la toma de medidas. Vamos a detallar cómo se interacciona con el interfaz al realizar cada una de estas acciones. Empezaremos por agregar un dispositivo.

Agregar dispositivo

Al pulsar sobre el botón “Añadir dispositivo”, se nos muestra la siguiente pantalla:

Figura 29: Selección de tecnología

La aplicación muestra las tecnologías de dispositivo que soporta el interfaz biométrico. Dado que nosotros sólo hemos implementado Bluetooth como tecnología soportada, sólo se nos muestra ésta. Al pulsar sobre la tecnología deseada, el interfaz biométrico se encargará de comprobar si está todo en orden para buscar dispositivos de dicha tecnología. En nuestro caso, al no estar Bluetooth activado, se nos comunica si deseamos habilitarlo, como se puede ver en la Figura 30.

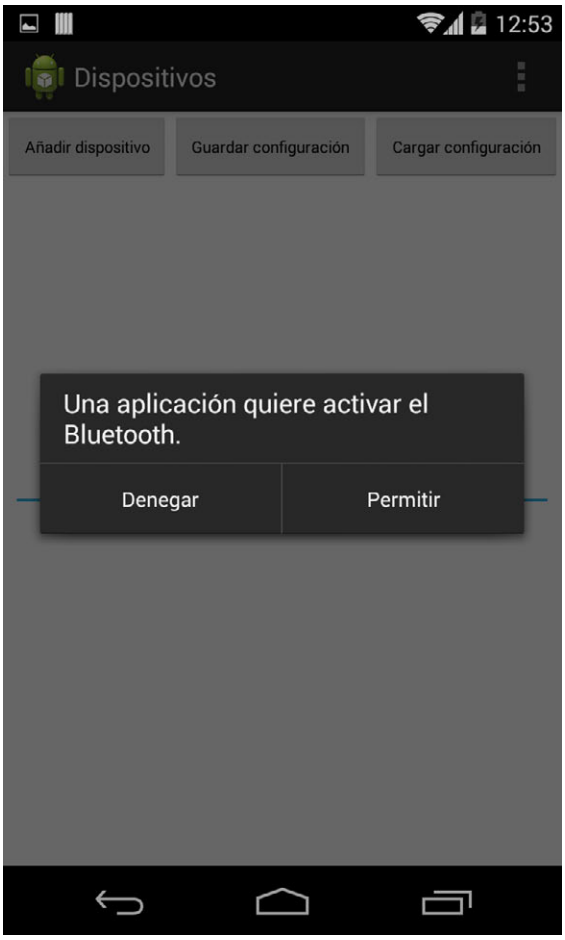

Figura 30: Petición de activación de Bluetooth

Una vez habilitado, comienza la búsqueda de dispositivos. Se nos mostrarán los dispositivos con la tecnología seleccionada que se hayan encontrado.

Finalmente, pulsamos el dispositivo deseado, tras lo cual se nos mostrará el diálogo de la Figura 31.

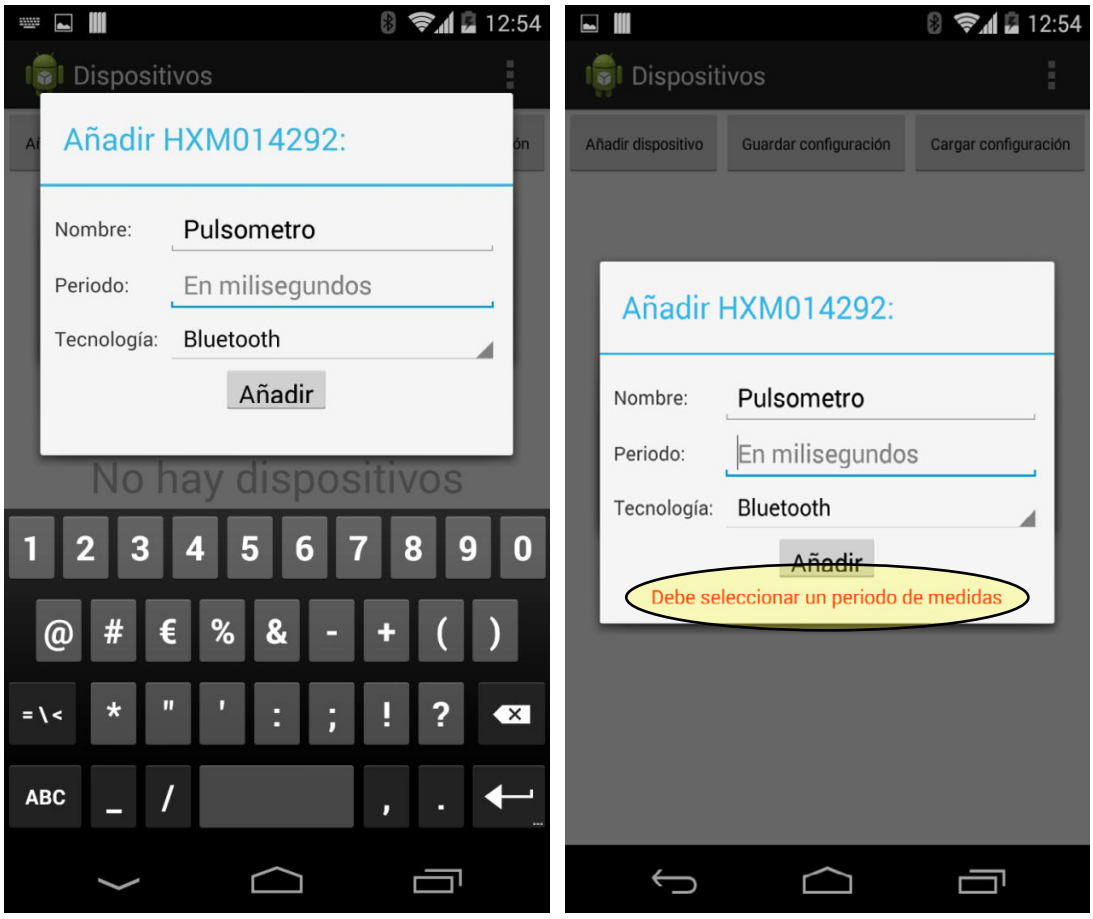

Figura 31: Diálogo de adición de dispositivo

En dicho diálogo, se nos pide configurar lo siguiente:

- Nombre: Un nombre para el dispositivo a agregar que sea reconocible fácilmente al ser leído por una persona.
- Periodo: El periodo, en milisegundos, para la toma de medidas. Es decir, se enviará una medida biométrica procedente de este dispositivo hacia el Smartphone cada vez que transcurra este periodo.
- Tecnología: La tecnología que utiliza este dispositivo.

Si algún dato introducido por el usuario no se acepta, por ejemplo un periodo de tiempo no válido, se mostrará un mensaje de error hasta que se haya introducido todo correctamente (ver Figura 31).

Editar dispositivo

Esta funcionalidad permite al usuario modificar algún parámetro de los dispositivos que tiene agregados actualmente. Para ello, lo que se debe hacer es pulsar sobre uno de los dispositivos que tenga, y le aparecerá un diálogo como el de la Figura 32.

Figura 32: Diálogo edición de dispositivo

Se puede apreciar que con la implementación actual sólo es posible modificar el periodo con el que se reciben datos del dispositivo. Para modificarlo, simplemente se necesita introducir otro valor y pulsar el botón “Atrás” del terminal. Si se detectan cambios, se actualizará el valor del dispositivo. En caso contrario se dejará como estaba.

Guardar/Cargar configuración

Para evitar hacer que el usuario deba introducir la misma configuración una y otra vez a la hora de hacer uso de sus dispositivos, se permite al usuario guardar la configuración de los dispositivos que tiene actualmente agregados. Se puede ver en la Figura 33 el aspecto que tiene este fichero.

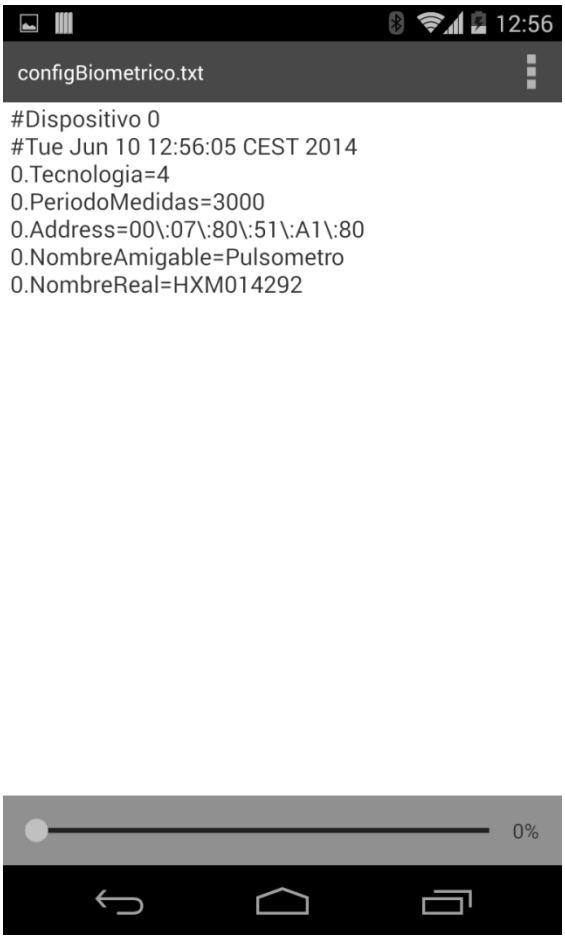

Figura 33: Fichero de configuración

Al igual que guardar, se nos ofrece la posibilidad de cargar la configuración guardada. Simplemente se utilizará un fichero de configuración previamente guardado y se mostrará la lista de dispositivos agregados al conector.

Conectar/Desconectar dispositivo

Por último, finalizamos con las no menos importantes funciones de conectarse o desconectarse al dispositivo. Éstas sirven básicamente para conectarse al dispositivo correspondiente y comenzar a tomar medidas de éste y desconectarse cuando así se solicite.

El programador de la aplicación tendrá que tener el driver del dispositivo ya desarrollado, de forma que a la hora de realizar medidas, el conector biométrico conozca cómo se debe proceder para comunicarse con el dispositivo. Si no fuese así, no sería posible recibir medidas.

Para realizar la conexión, se debe pulsar sobre el dispositivo correspondiente en la lista de dispositivos agregados, vista en la Figura 28. Posteriormente, en el diálogo que nos aparece, pulsaremos en el botón “Conectar”.

A partir de éste momento, se iniciará el proceso de conexión. Como se vio anteriormente en el apartado 3.2.2.1., que describe la especificación del interfaz

biométrico, se nos podían presentar dos tipos de eventos, de sistema y de dispositivo. En el proceso de conexión surgirán eventos de sistema que tendremos que procesar para poder saber qué está ocurriendo en todo momento mientras se realiza la conexión.

Si intentamos conectarnos a un dispositivo que no está habilitado o el Smartphone del usuario no puede comunicarse, se nos avisará de ello. En la Figura 34 se puede ver un ejemplo de esto. Hemos intentado conectarnos a un dispositivo por Bluetooth, pero debido a que el terminal no lo tenía habilitado, se ha mostrado un mensaje avisando de ello.

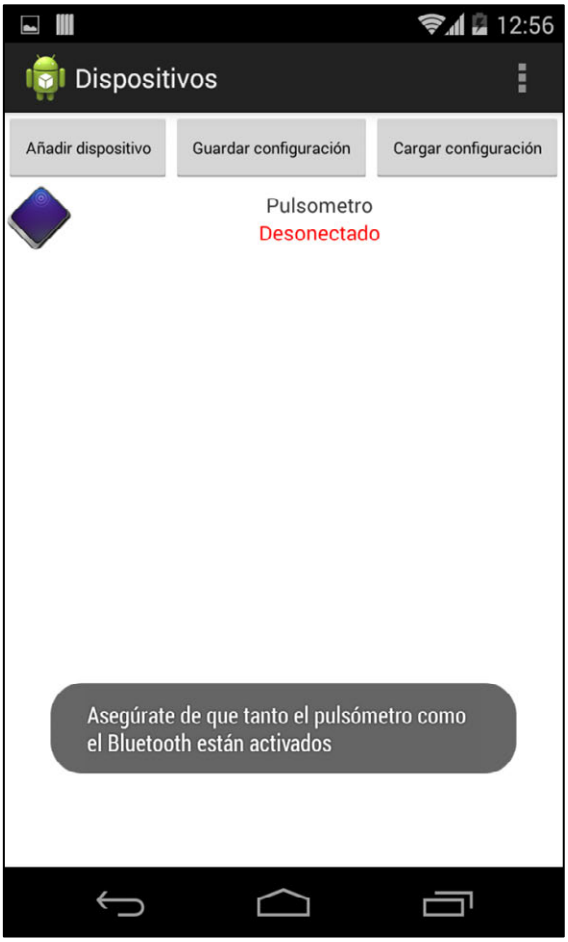

Figura 34: Aviso al usuario

Cuando la conexión es satisfactoria, se advertirá al usuario de ello y se comenzará a tomar medidas automáticamente. En la siguiente figura se puede observar el listado de dispositivos con indicación de aquéllos que están conectados.

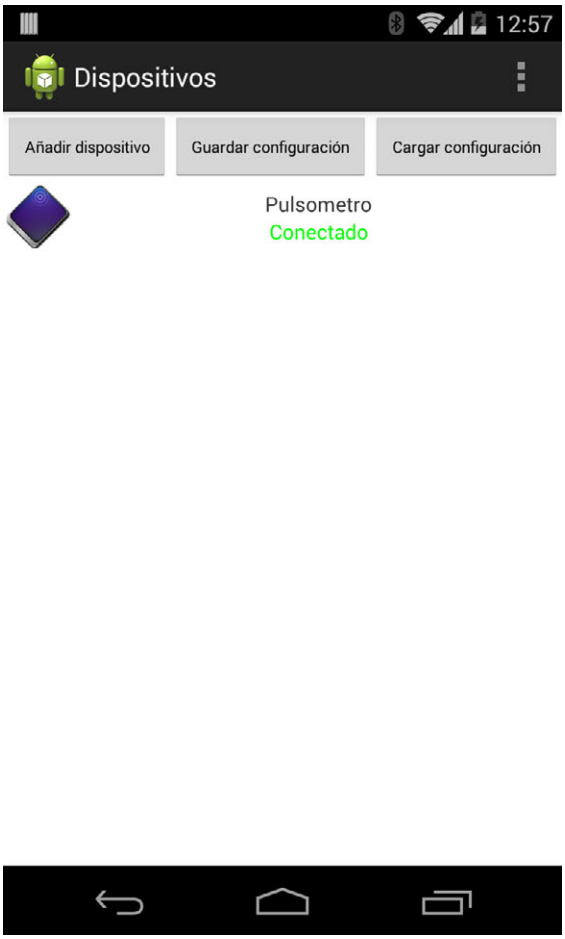

Figura 35: Lista de dispositivos con dispositivo conectado

Pantalla Medidas

Pasemos ahora a la pantalla de medidas. Como comentamos anteriormente, es aquí donde el usuario podrá ver las medidas que se encuentren almacenadas en la base de datos. Puede verse en la Figura 36.

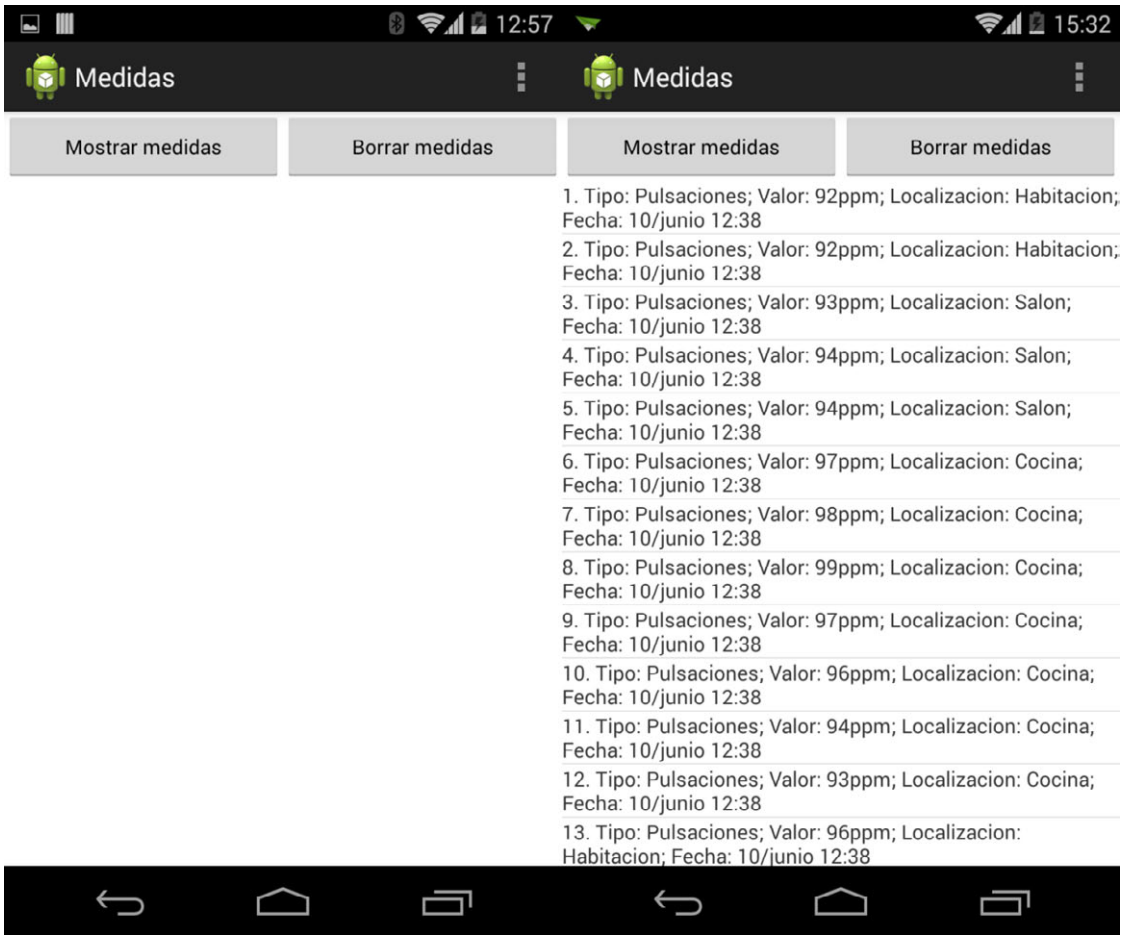

Figura 36: Pantalla de medidas

Cuando el usuario solicita las medidas, éstas se recuperan haciendo uso del interfaz de base de datos. El programador se encarga de hacer uso de las funciones que ofrece el interfaz para obtener las medidas y se encarga de formatearlas para mostrarlas en pantalla de forma que sean entendibles.

Pantalla Gráfica

Para finalizar, explicaremos el funcionamiento de la última pantalla de la aplicación, de la cual se puede ver el aspecto que presenta en la Figura 37.

Figura 37: Pantalla de configuración de la gráfica

Esta pantalla presenta dos fragmentos: Uno dedicado a la configuración de los parámetros de la gráfica, y otro para la representación de ésta.

En lo que respecta a la parte de la configuración de la gráfica, se presentan una serie de parámetros configurables por el usuario, que son los siguientes:

- Medida biométrica: El tipo de medida biométrica que se quiere representar.
- Medida ambiental: El tipo de medida ambiental que se quiere representar.
- Desde – Hasta: Intervalo de fecha en el que se quiere representar la gráfica.

Una vez seleccionados los parámetros y pulsando en el botón de crear, la aplicación procederá, mediante el interfaz B.D., a solicitar un listado de medidas biométricas que existan en el intervalo de tiempo configurado. En caso de no encontrarse medidas biométricas, se informará al usuario y no se dibujará ninguna gráfica, ya que el objetivo de esta aplicación es poder relacionar una serie de medidas biométricas con medidas ambientales. Sin embargo, en caso de sí existir medidas biométricas, se procederá a continuación a solicitar las medidas ambientales correspondientes.

En este punto, se hará uso del interfaz ambiental, para solicitar dichas medidas. Como las biométricas tienen una fecha y localización en las que fueron tomadas, lo que la aplicación hace es obtener según fecha y localización. Esto se hace así porque queremos mostrar sólo medidas ambientales de los momentos en los que el usuario estaba tomando medidas biométricas y en el lugar donde las estaba tomando. No tendría sentido relacionar una medida tomada en la cocina (por ejemplo) con una tomada en el salón, aunque hubieran sido tomadas al mismo tiempo.

Así, una vez tenemos las medidas, tanto biométricas como ambientales, procedemos a la representación, haciendo uso de la funcionalidad que nos ofrece el interfaz gráfico. A continuación, en la Figura 38, se muestra un diagrama de todo el proceso comentado, para una mejor visualización:

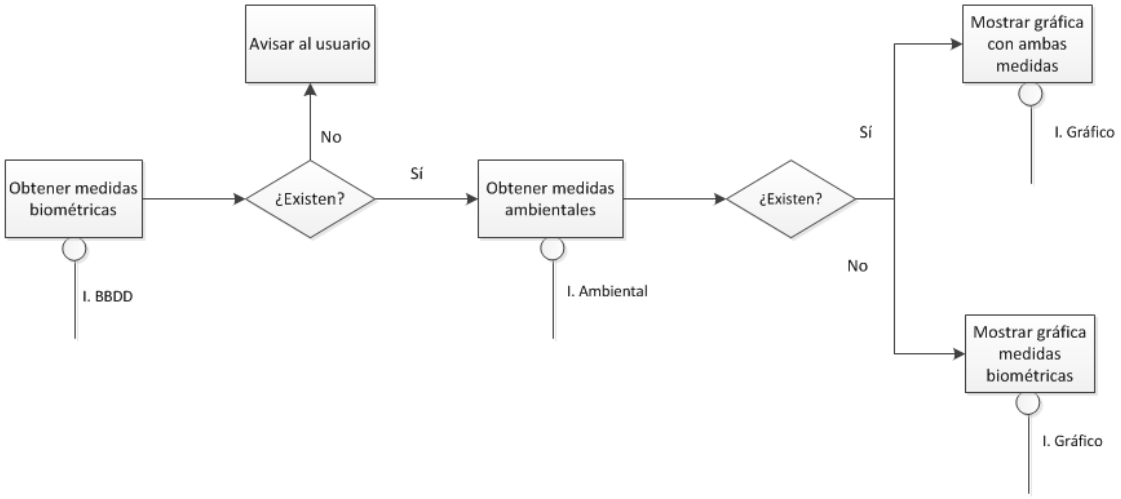

Figura 38: Diagrama de creación de la gráfica

En el diagrama se muestran, en los procesos, el interfaz correspondiente del que hacen uso para realizar la función indicada.

3.3.3. Implementación de los módulos

Procederemos ahora a ver cómo se han implementado los distintos módulos que ofrecen las interfaces diseñadas, haciendo una descripción de cada una de las clases que los componen. Se ha implementado un paquete de clases para cada uno de estos módulos.

3.3.3.1. Diagrama de clases de los módulos implementados

Vimos los diagramas de clases de los interfaces diseñados, representando lo que deberían ofrecer parar cumplir las funcionalidades mínimas que se quieren lograr con ellos. En este apartado se muestran los diagramas de los módulos implementados, los cuales ofrecen todas las funciones que el interfaz requiere.

Diagrama de clases del Módulo de obtención de medidas biométricas

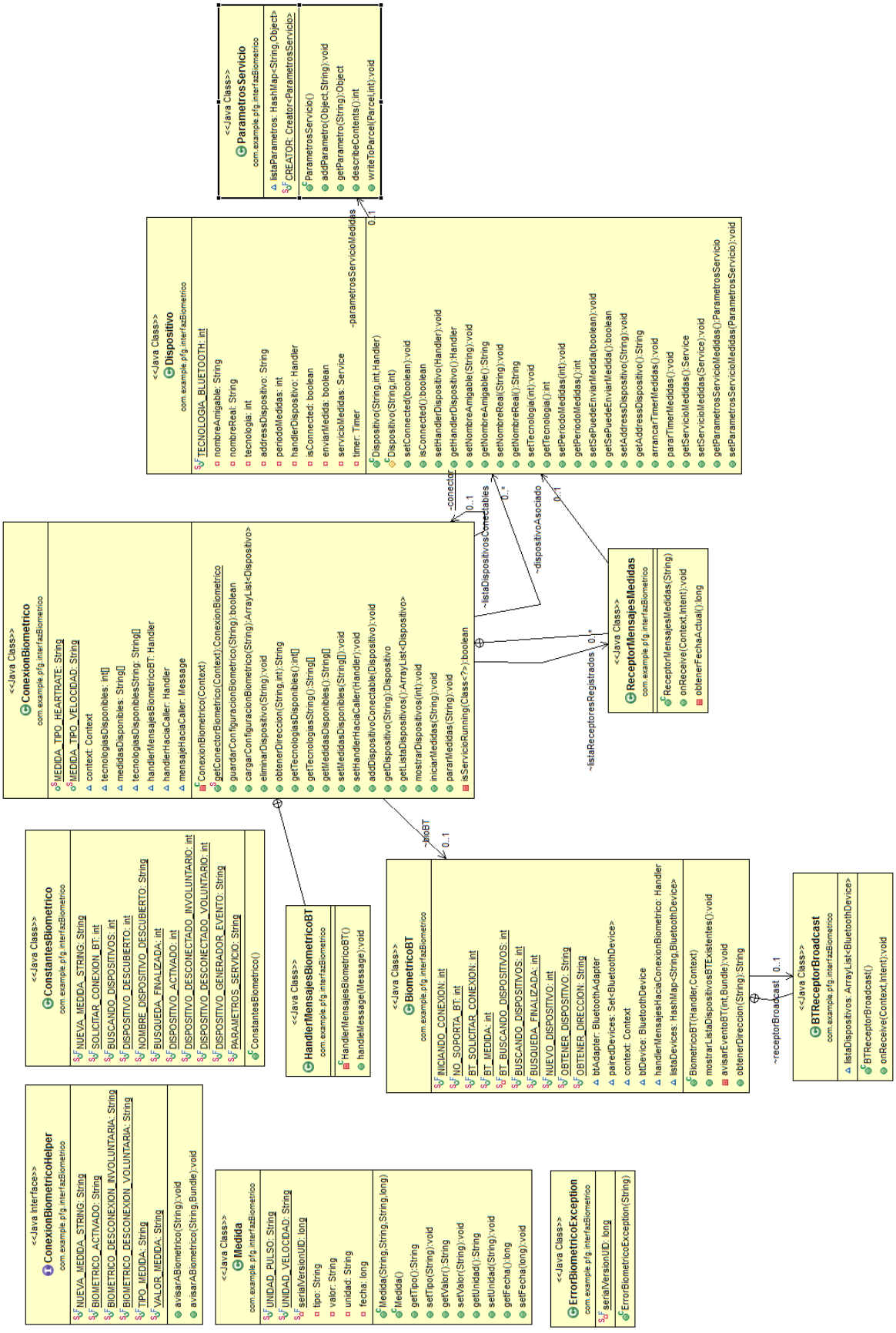

Figura 39: Diagrama de clases del módulo de obtención de medidas biométricas

Diagrama de clases del Módulo de almacenamiento y recuperación de medidas biométricas

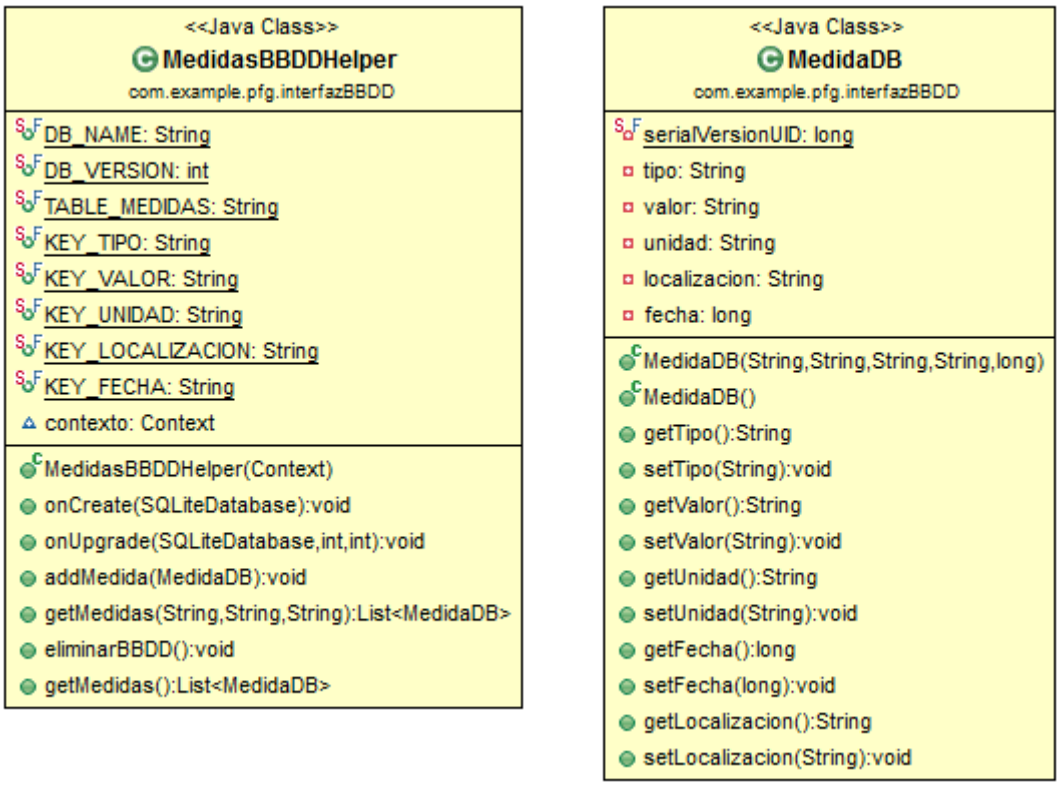

Figura 40: Diagrama de clases del módulo de almacenamiento y recuperación de medidas biométricas

Diagrama de clases del Módulo de obtención de medidas ambientales

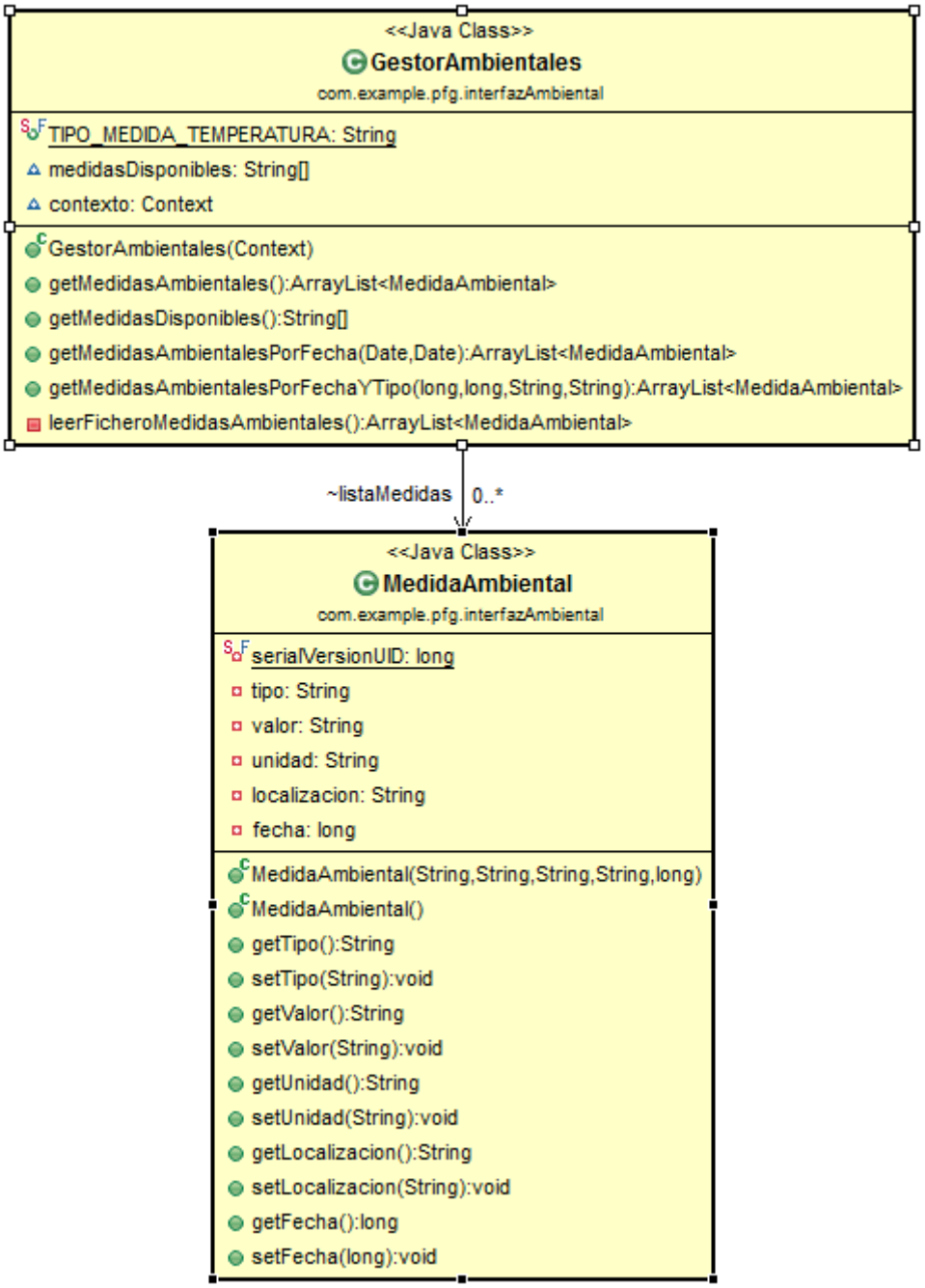

Figura 41: Diagrama de clases del módulo de obtención de medidas ambientales

Diagrama de clases del Módulo de representación gráfica de medidas

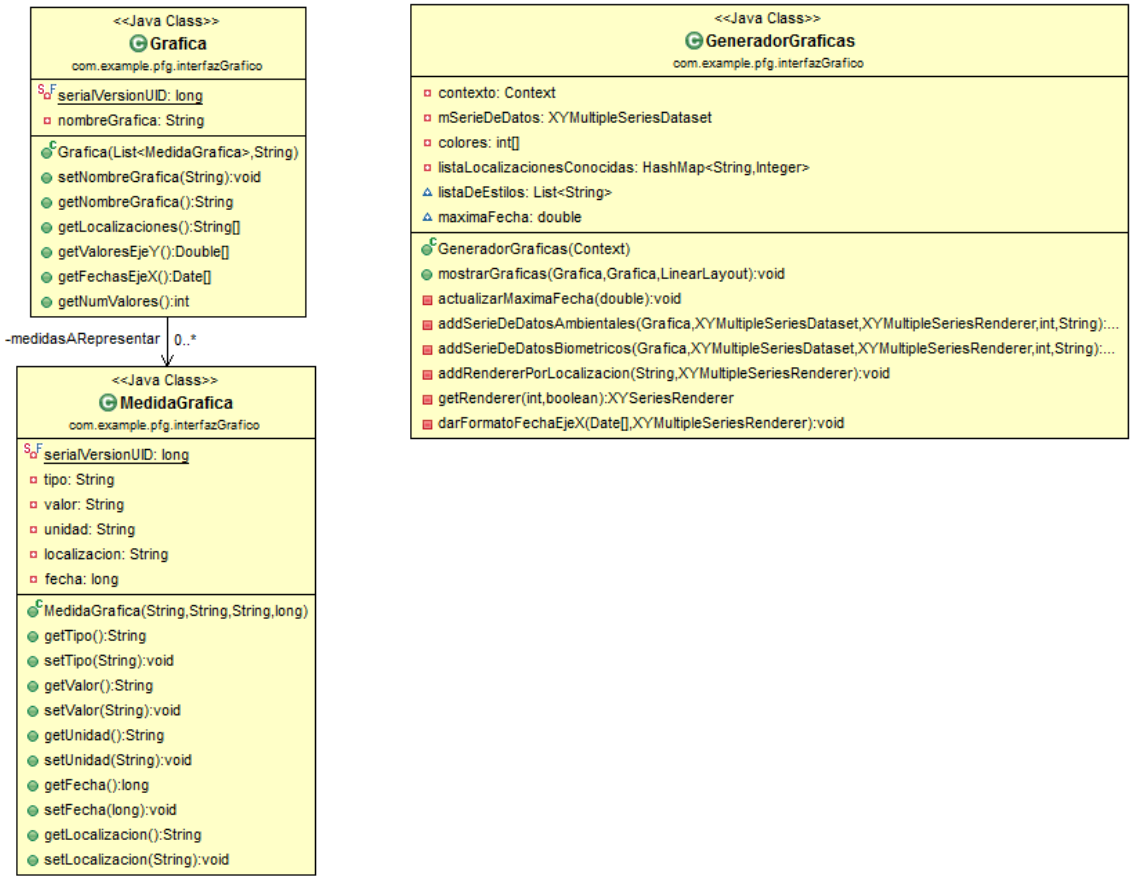

Figura 42: Diagrama de clases del módulo de representación gráfica de medidas

3.3.3.2. Módulo de obtención de medidas biométricas

En la Figura 43 se muestran las clases de la que está compuesto el módulo de obtención de medidas biométricas.

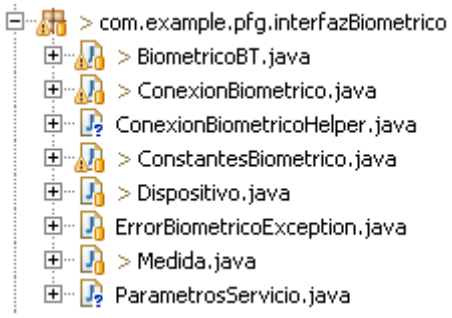

Figura 43: Lista de clases del módulo de obtención de medidas biométricas

Dispositivo.java

Esta clase representa el modelo de dispositivo utilizado. Contiene todos los parámetros necesarios que se esperan de un dispositivo para ser utilizado por la clase (ver Figura 44).

```
private String nombreAmigable;
private String nombreReal;
private int tecnologia;
private String addressDispositivo;
private int periodoMedidas;
private Handler handlerDispositivo;
private boolean isConnected;
private boolean enviarMedida;
private Service servicioMedidas;
private ParametrosServicio parametrosServicioMedidas;
private Timer timer;
```

Figura 44: Variables de la clase Dispositivo.java

En la figura anterior se pueden observar los siguientes parámetros.

- `nombreAmigable`: Un nombre que da el programador para identificar fácilmente un dispositivo concreto.
- `nombreReal`: El nombre de fábrica qué tiene el dispositivo.
- `tecnologia`: Entero que define la tecnología que utiliza el dispositivo. La misma clase ofrece una serie de constantes con las tecnologías soportadas por el momento.
- `addressDispositivo`: La dirección física del dispositivo.
- `periodoMedidas`: El periodo tras el cual se toma una medida.
- `handlerDispositivo`: Manejador del dispositivo para comunicar al programador, a través de éste, eventos relacionados con el dispositivo, como pueden ser las medidas.
- `parametrosServicioMedidas`: Los parámetros que el programador pueda necesitar pasar al servicio que actuará de driver.

- `isConnected`: Variable para conocer si el dispositivo está conectado y tomando medidas o no.

Por último hay dos parámetros que conviene explicar con más profundidad, que son `enviarMedida` y `timer`. En un caso ideal, el objeto encargado de solicitar medidas al dispositivo debería pedir una medida cuando pasa el periodo de medida determinado. Sin embargo, debido a que el dispositivo biométrico concreto con el que se han realizado las pruebas no permitía pedirle medidas en instantes determinados, sino que difundía las medidas que va tomando cada un periodo de tiempo determinado (que tampoco es configurable), hemos tenido que simular dicha petición. Es por ello que se han declarado estas dos variables. Se van a estar recibiendo medidas constantemente, pero sólo nos interesan aquellas que recibamos cuando pase el tiempo configurado. Por ello, se tiene una variable que indica si se puede enviar una medida al programador o no, y un *timer* que, cuando pasa el periodo de tiempo configurado, pone la variable `enviarMedida` a verdadero.

De esta forma, cuando se reciben medidas, se comprueba si esta variable es verdadera. En caso afirmativo, se envía la medida al programador. En caso negativo, se ignora la medida, hasta que recibamos una medida y la variable esté con valor verdadero.

ConexionBiometricoHelper.java (Interfaz)

Esta interfaz ha sido creada con el propósito de establecer qué debe realizar el programador del driver del dispositivo para la comunicación entre el conector biométrico y el propio dispositivo.

El driver está implementado en forma de servicio. Un servicio es un componente de la aplicación que puede realizar operaciones por un largo periodo de tiempo y que no ofrece interfaz gráfica de usuario. La toma de medidas se realizará en segundo plano, para permitir al usuario continuar utilizando la aplicación.

Por tanto, el programador debe programar el driver en forma de servicio, y que además implemente esta interfaz. La interfaz contiene dos métodos (denominados ambos `avisarABiometrico`) con el único propósito de informar de un determinado evento relacionado con el dispositivo biométrico al conector. Se ofrecen dos métodos dado que existe un caso en el que el evento puede ser una medida procedente del dispositivo, y, por tanto, se debe comunicar el evento junto a la medida. Para realizar correctamente la comunicación de los eventos, se explica, en la documentación de esta interfaz adjuntada en el anexo de este proyecto, que el usuario debe hacer uso de los denominados LocalBroadcastManager que ofrece Android, de forma que se enviará un broadcast a nivel de aplicación (por lo que no escapa al resto del sistema) con el evento producido.

Como los eventos no son conocidos a priori por el usuario, se ofrecen constantes indicando cuáles de ellos son los que el conector espera recibir en algún momento.

ConexionBiometrico.java

Es la clase principal del paquete. Es la que manejará el programador a la hora de realizar la gestión de los dispositivos biométricos, como añadirlos, conectarse a ellos, guardar o cargar sus configuraciones.

La particularidad que tiene esta clase es que sigue un patrón *Singleton* (ver Figura 41). Este patrón está diseñado para restringir la creación de objetos pertenecientes a una clase. Nosotros queremos que en todo momento sólo exista una instancia de la clase ya que concebimos que exista un solo gestor para los dispositivos y no varios.

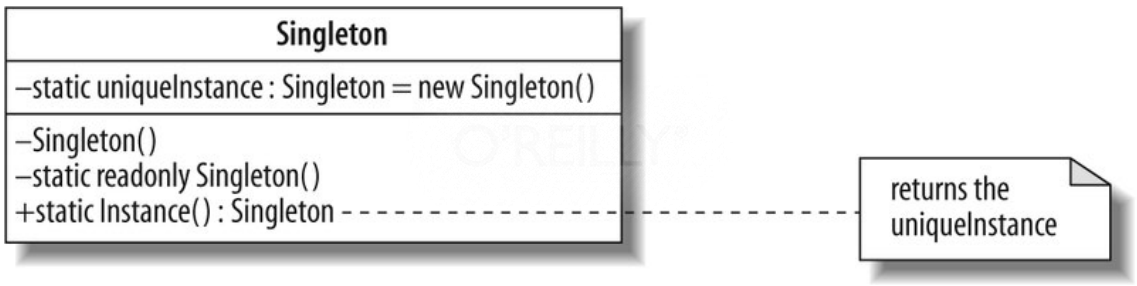

Figura 45: Estructura general de una clase Singleton [27]

Como hemos comentado, la clase permite añadir un número de dispositivos indefinido, para lo cual se ha definido un mapa que almacene pares clave-valor, de manera que el programador definirá la clave para el objeto (dispositivo) que vaya a almacenar.

Para la comunicación entre programador y conector, se mencionó en la especificación del interfaz biométrico que es necesario asociar al conector un manejador de eventos de sistema. Nosotros hemos realizado esto haciendo uso de la clase Handler que proporciona Android. Así, la persona encargada de utilizar este módulo, tendrá que implementar una clase que herede de la clase Handler, crear una instancia, y usar el método `setHandlerHaciaCaller` para indicarle la referencia a dicha instancia. A partir de este momento, la comunicación de eventos de sistema se podrá realizar satisfactoriamente.

Una parte fundamental de esta clase es cómo se procede a la hora de realizar la toma de medidas, por lo que vamos a profundizar en ello en los siguientes párrafos:

De entrada, será necesario que exista de manera previa el denominado driver (el servicio con la interfaz ConexionBiometricoHelper implementada) del dispositivo (anteriormente descrito).

Suponiendo que el driver del dispositivo existe, cuando se hace una llamada al método encargado de iniciar las medidas, lo primero que se realiza es registrar un receptor de broadcasts que se produzcan en la aplicación.

Cuando se realiza el registro de este receptor, lo que estamos haciendo es permanecer a la escucha de broadcasts lanzados por el servicio, y, cuando llegue uno, actuar en consecuencia.

Posteriormente se lanza el driver que se comunicará con el dispositivo biométrico para informar de las medidas. Permanecerá en segundo plano e informará en todo momento de cualquier evento relacionado con el dispositivo que pueda ocurrir, además de las medidas. Por ejemplo, si nos hemos desconectado de éste involuntariamente. Es por ello que es importante que el driver esté correctamente implementado, teniendo en cuenta todos los eventos que pueden ocurrir y sean relevantes para el programador. En caso de omitir alguno de estos eventos, podría no funcionar correctamente la aplicación desarrollada con estas interfaces.

Para finalizar, arrancamos el Timer de medidas del dispositivo. Como comentamos anteriormente, el dispositivo concreto que hemos usado en este proyecto no permite que se le configure un periodo para que difunda medidas cada determinado tiempo, ni permite que se le pida una medida en el momento deseado. Así, al arrancar este Timer, lo que haremos es tener en cuenta las medidas que nos mande el driver solamente cuando haya transcurrido el periodo configurado.

En la Figura 46 podemos ver un diagrama de lo explicado, para verlo más fácilmente.

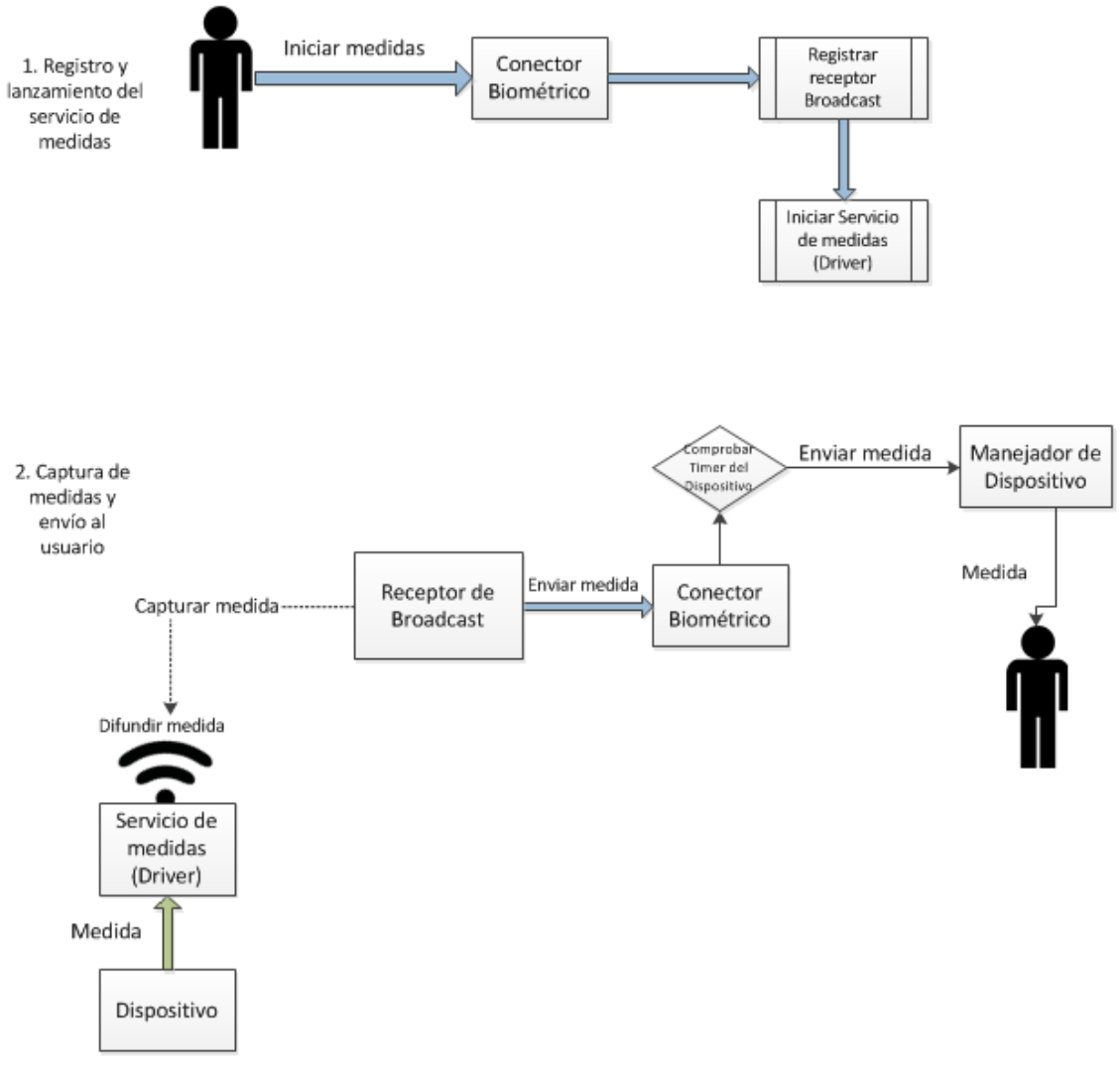

Figura 46: Diagrama de petición de medidas

BiometricoBT.java

Cuando describíamos el funcionamiento del interfaz biométrico, vimos que existen unos gestores internos para manejar diferentes tecnologías que pueden presentar los dispositivos.

La clase ConectorBiometrico.java está programada de forma que, cuando se va a realizar alguna operación sobre un dispositivo, primero se comprueba qué tecnología utiliza este para adaptar las operaciones a dicha tecnología. Por ejemplo, cuando se quiere obtener la dirección de un dispositivo, se comprueba qué tecnología utiliza y posteriormente se utiliza el gestor de dicha tecnología para recibir la dirección.

En este caso concreto, BiometricoBT.java es la clase encargada de gestionar funciones relacionadas con tecnología Bluetooth. Funciones como obtener dirección Bluetooth de un dispositivo o mostrar dispositivos del entorno qué utilicen Bluetooth se llevan a cabo haciendo uso de esta clase.

ConstantesBiometrico.java

Esta clase ha sido creada con el único objetivo de ofrecer al programador una serie de constantes que vaya a necesitar para eventos que sucedan.

ErrorBiometricoException.java

Es una excepción desarrollada para informar de errores que surjan en la clase ConectorBiometrico.java

Medida.java

Esta clase modela un objeto medida que utiliza el paquete biométrico.

Cuando se obtienen medidas del dispositivo, al programador le llegarán instancias de esta clase, representándolas. En la figura siguiente se observan los atributos de dicha clase:

```
private String tipo, valor, unidad;
private long fecha;
```

Figura 47: Variables de la clase Medida.java

- Tipo: El tipo de medida. Puede ser pulsación, velocidad, distancia, etc.
- Valor: El valor de la medida.
- Unidad: Unidad en la que está expresado el valor.
- Fecha: Fecha en la que tomada la medida. Está en formato POSIX, que es un sistema para la descripción de instantes de tiempo. Se define como la cantidad de segundos transcurridos desde la medianoche UTC del 1 de Enero de 1970. [**28**]

ParametrosServicio.java

Como hemos comentado, el programador debe proporcionar un servicio que implemente la interfaz ConexionBiometricoHelper.java.

Se dan casos en los que este servicio requiere de parámetros adicionales que el programador querría poder pasarle. Es por esto que se ha creado esta clase.

El programador puede añadir los parámetros que le sean necesarios, dando un nombre para poder identificarlos. A la hora de recuperarlos, deberá hacer un *casting* de éstos al tipo esperado, ya que siempre se devolverán con tipo Object.

3.3.3.3. Módulo de almacenamiento y recuperación de medidas biométricas

La Figura 48 muestra las clases pertenecientes a este módulo.

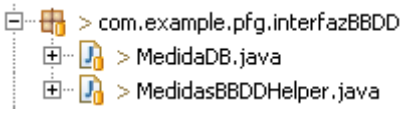

Figura 48: Lista de clases del módulo de almacenamiento y recuperación de medidas biométricas

MedidasBBDDHelper.java

Esta clase encapsula las funciones necesarias para realizar una base de datos y añadirle o extraerle datos. También permite eliminar la base de datos cuando así se considere necesario.

Está creada de forma que el programador no tenga siquiera que preocuparse de realizar la conexión con B.D., ya que se realizará de forma automática cada vez que se requiera.

Se utiliza la tecnología SQLite que proporciona Android para manejar las bases de datos. Cuando el programador desee guardar una medida, deberá crear un objeto de tipo MedidaDB (descrito a continuación) y llamar al método `addMedida`.

MedidaDB.java

Al igual que con el interfaz biométrico, el interfaz base de datos hace uso de un modelo de medida propio, al que se deberán adaptar las medidas que se vayan a almacenar. Este modelo incluye tanto indicación del instante como de la localización relativa a la medida en cuestión.

3.3.3.4. Módulo de obtención de medidas ambientales

La Figura 49 muestra las clases pertenecientes a este módulo.

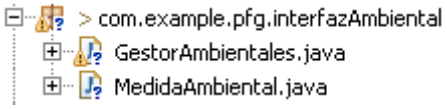

Figura 49: Lista de clases del módulo de obtención de medidas ambientales

El módulo de obtención de medidas ambientales está implementado de una manera análoga al módulo de almacenamiento y recuperación de medidas biométricas.

Para simular la existencia de un servidor al que se realicen las peticiones de las medidas ambientales, se ha procedido a crear un fichero XML con una serie de medidas preconfiguradas, que leemos para devolver al usuario una lista de medidas con igual formato a la que se recibiría al realizar la petición de medidas a la base de datos local.

El fichero XML sigue el siguiente *schema*:

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="listaMedidas">
      <xs:complexType>
          <xs:sequence>
              <xs:element ref="medida" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
      </xs:complexType>
    </xs:element>
  <xs:element name="medida">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="xs:string" name="tipo"/>
        <xs:element type="xs:string" name="valor"/>
        <xs:element type="xs:string" name="unidades"/>
        <xs:element type="xs:string" name="localizacion"/>
        <xs:element type="xs:string" name="fecha"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figura 50: Schema para validar documentos XML

Este *schema* se ha creado con la idea de que, cuando se genere un fichero XML con medidas preconfiguradas, debamos validarlo para así asegurarnos de que existen todos los campos necesarios que componen una medida.

En la Figura 51 podemos ver un ejemplo de fichero XML formado:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<listaMedidas>
    <medida>
        <tipo>Temperatura</tipo>
        <valor>22</valor>
        <unidades>ºC</unidades>
        <localizacion>Cocina</localizacion>
        <fecha>1400332311</fecha>
    </medida>
    <medida>
        <tipo>Temperatura</tipo>
        <valor>25</valor>
        <unidades>ºC</unidades>
        <localizacion>Cocina</localizacion>
        <fecha>1400332312</fecha>
    </medida>
    <medida>
        <tipo>Temperatura</tipo>
        <valor>23</valor>
        <unidades>ºC</unidades>
        <localizacion>Cocina</localizacion>
        <fecha>1400332313</fecha>
    </medida>
    <medida>
        <tipo>Temperatura</tipo>
        <valor>25</valor>
        <unidades>ºC</unidades>
        <localizacion>Cocina</localizacion>
        <fecha>1400332314</fecha>
    </medida>
```

Figura 51: Estructura del fichero XML generador de medidas ambientales

Los elementos son los campos pertenecientes a un objeto medida de tipo MedidaAmbiental.java. Aunque en este caso es idéntico al modelo de medida del módulo de obtención de medidas biométricas consideramos que, debido a la independencia que debe existir entre los módulos, cada uno debe tener las medidas formateadas según sus necesidades, puesto que habrá casos en los que se requiere una estructura de datos que difiera de lo necesitado por otro módulo.

3.3.3.5. Módulo de representación gráfica de medidas

La Figura 52 muestra las clases pertenecientes a este módulo.

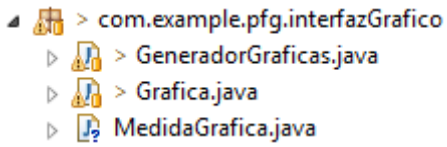

Figura 52: Lista de clases del módulo de representación gráfica de medidas

MedidaGrafica.java

Como se comentó en módulos anteriores, cada uno de ellos tiene su propio modelo de medida debido a la independencia que debe existir entre todas. Por tanto, el interfaz gráfico utiliza la clase MedidaGrafica.java para definir cómo debe ser una medida para poder realizar las representaciones de manera adecuada.

Grafica.java

La clase Grafica.java tiene como objetivo crear un modelo de gráfica que será utilizado a la hora de realizar su representación.

Antes de poder representar medidas, es necesario que el programador cree una instancia de esta clase y le pase los parámetros necesarios, que no serán más que la lista de medidas que se pretende que tenga la gráfica junto a un nombre identificativo de la misma.

Internamente, lo que la clase realiza es obtener los valores de las medidas así como los instantes de tiempo en los que fueron tomadas, separándolos en dos listas diferentes.

Contiene otros métodos de los que hará uso la clase encargada de representar la gráfica en pantalla, GeneradorGraficas.java.

- **GeneradorGraficas.java**

Ésta es la clase principal del paquete correspondiente al módulo de representación gráfica.

Como se mencionó en el apartado 3.3.1., se ha procedido a utilizar la librería AChartEngine para realizar las gráficas.

Como características destacables de esta librería para el propósito que perseguimos, están la posibilidad de mostrar dos ejes de magnitudes diferentes, personalizar las etiquetas del eje de tiempo, y mostrar tantas gráficas como sea necesario. En nuestro caso, el hecho de poder establecer dos ejes en la misma gráfica nos permite mostrar dos tipos de medidas diferentes sin mayor dificultad.

Por tanto, pasemos a detallar cómo funciona el interfaz.

Anteriormente mencionamos la clase Grafica.java, la cual hace de modelo que utilizará la clase GeneradorGraficas.java para mostrarlas.

El programador que haga uso de esta clase, utilizará simplemente un método, denominado `mostrarGraficas` cuyos parámetros serán las gráficas que desee representar en pantalla y el layout en el cual se va a representar. Una vez se llama a dicho método, el funcionamiento es el que se describe en los siguientes párrafos:

En primer lugar, se comprueba si el usuario ha pasado una o dos gráficas al método. Esto se realiza para ver si debemos dibujar uno o dos ejes en pantalla, y, además, para

saber si sólo se van a representar medidas biométricas o tanto biométricas como ambientales. En caso de representar ambas, las ambientales se muestran en forma de barra y las biométricas en forma de línea, para así diferenciarlas.

La librería AChartEngine presenta dos objetos principales para crear el gráfico: uno de ellos para almacenar las series de datos, que son conjuntos de parejas con los valores para el eje X y el eje Y; y otro denominado *renderer*, que será el que se encargue del aspecto que presenta la gráfica en sí.

Por tanto, una vez hemos comprobado el número de gráficas, se pasa a dar formato a los datos para almacenarlos en los objetos de series de datos. Un objeto de serie de datos consiste en una lista con una serie de valores X-Y de tipo `double`, ya que así está diseñado. Como valor X, almacenaremos la fecha en formato POSIX. Como valor Y, almacenamos el valor de la medida.

La complejidad de la representación gráfica se presenta a la hora de diferencias las medidas según la zona en la que se encontrase el usuario. En la implementación realizada en este proyecto se ha decidido diferencias las zonas mediante colores distintos. Un objeto de tipo Grafica.java contiene además de los valores de las medidas y los instantes en los que son tomadas, la lista de localizaciones donde fueron tomadas.

La librería AChartEngine no permite colorear una única serie de datos con diferentes colores, sino que cada serie tiene asociado un color que nosotros definamos. Para solucionar esto, lo que hemos realizado es crear tantas series de datos como localizaciones se vayan presentando al ir recorriendo la lista de medidas de la gráfica. Es decir, si, por ejemplo, en un determinado punto del recorrido, obtenemos una medida tomada en una habitación, posteriormente otra tomada en el salón, y posteriormente otra tomada en la habitación, crearíamos tres series de datos. Sin embargo, si obtenemos tres medidas seguidas con la misma localización, sólo crearíamos una.

Una vez solucionado el coloreado de la gráfica, habría que dar formato al eje X. AChartEngine, por defecto, pone como etiquetas en el eje X los valores X que hayamos agregado a las series de datos. Esto quiere decir que, cuando se nos muestre la gráfica, se mostrarán fechas en formato POSIX, lo cual es poco amigable. Por suerte, AChartEngine da multitud de opciones a la hora de personalizar las gráficas. Como comentamos anteriormente, el objeto *renderer* es el encargado de personalizar el diseño que se va a mostrar. Entre sus posibilidades, ofrece un método para reemplazar un valor del eje X por una cadena de texto introducida por nosotros. Por tanto, lo que hemos hecho es sustituir el instante en formato POSIX por una fecha legible.

La gráfica que finalmente se muestra, se puede observar en las figuras 66 y 67 del apartado 4.3.

3.3.4. Implementación de la aplicación móvil

En la Figura 53 pueden observarse las clases que componen la aplicación móvil. Las hemos resaltado en varios colores según qué parte de la aplicación constituyen.

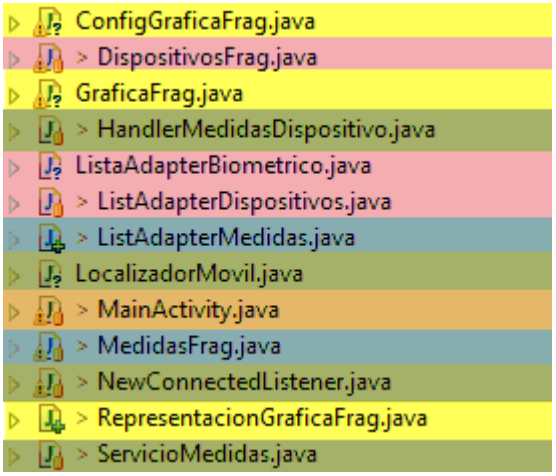

Figura 53: Clases de la aplicación móvil

- En primer lugar, la actividad principal (y la única en la aplicación) se puede ver en naranja con el nombre de MainActivity.java. La aplicación está diseñada haciendo uso de Fragments [**29**], por lo que no se ha considerado realizar más que una actividad.

- A continuación, la pantalla de dispositivos está construida con la clase DispositivosFrag.java, resaltada en rosa. No obstante, para realizar las listas de dispositivos encontrados a la hora de añadir nuevos dispositivos o para mostrar la lista de dispositivos procedentes del conector biométrico, se han realizado dos clases más, que son ListAdapterBiometrico.java y ListAdapterDipositivos.java.

- La pantalla de Medidas la compone MedidasFrag.java, resaltada en azul, y ListAdapterMedidas.java. La primera de ellas contiene todas las operaciones de la pantalla de las medidas en sí, mientras que la segunda es para dar formato a la lista de medidas que se muestra en pantalla.

- La última pantalla, la pantalla gráfica, está formada por tres fragments. El principal, GraficaFrag.java, es un contenedor para incorporar otros dos fragments, el dedicado a configurar los parámetros de la gráfica (ConfigGraficaFrag.java) y el dedicado a mostrar la gráfica resultante (RepresentacionGraficaFrag.java). Resaltados en amarillo.

- Por último, están las clases auxiliares, resaltadas en verde. En primer lugar, la clase HanlderMedidasDispositivo.java, será el manejador de eventos de dispositivo, de la cual se creará una instancia nueva por cada dispositivo que se añada. LocalizadorMovil.java se usa para simular la posición del usuario mientras está

tomando medidas. Y, para finalizar, ServicioMedidas.java y NewConnectedListener.java forman el servicio que tomará las medidas, el denominado driver que se comentó en el apartado 3.3.3.1, en la descripción de la interfaz ConexionBiometricoHelper.java.

3.3.4.1. Actividad principal (MainActivity.java)

Esta clase es la encargada de crear el menú de navegación y presentar los diferentes fragments en pantalla según la opción escogida. El menú está implementado como un Navigation Drawer [**30**]. El código de carga de los fragments puede verse en la Figura 54.

```
private class DrawerItemClickListener implements ListView.OnItemClickListener{
    @Override
    public void onItemClick(AdapterView parent, View view, int position, long id) {
            crearFragmentMenu(position);
    }

}

public void crearFragmentMenu(int position){
    FragmentManager fm = getSupportFragmentManager();
    //Determinamos el elemento de la lista que se ha pulsado y creamos un Fragment.
    switch(position){
        case 0: //Dispositivos
            fm.beginTransaction().replace(R.id.content_frame, new DispositivosFrag()).commit();
            break;
        case 1: //Medidas
            fm.beginTransaction().replace(R.id.content_frame, new MedidasFrag()).commit();
            break;
        case 2: //Gráfica
            fm.beginTransaction().replace(R.id.content_frame, new GraficaFrag()).commit();
            break;
    }

    mListaMenu.setItemChecked(position, true);
    //Seleccionamos el título que aparecerá en pantalla
    getActionBar().setTitle(mTitulos[position]);
    //Cerramos el menú
    mDrawerLayout.closeDrawer(mListaMenu);
}
```

Figura 54: Carga de fragments en MainActivity.java

Como se puede ver el menú tiene un listener para que, al hacer click en cualquiera de sus elementos, se llame al método `crearFragmentMenu` y cargue el fragment correspondiente.

3.3.4.2. Clases de la Pantalla Dispositivos

La clase que forma esta pantalla es Dispositivos.java. En ella se comienza obteniendo la instancia de conector biométrico. Como vimos, éste sigue un patrón *Singleton*, por tanto da igual en qué parte de la aplicación la obtengamos que la instancia siempre será la misma.

Una vez obtenemos la instancia, lo que hacemos es indicarle a esta cuál va a ser nuestro handler. Este handler será el encargado de manejar cualquier evento que nos tenga que comunicar el conector biométrico (evento que sea de sistema, no de dispositivo).

```
conectorBiometrico = ConexionBiometrico.getConectorBiometrico(getActivity());
conectorBiometrico.setHandlerHaciaCaller(handlerMensajesConector);
```

Figura 55: Código de obtención de Conector y selección de Handler

Nuestro handler (como vemos en la figura 55, `handlerMensajesConector`) está implementado como se muestra en la Figura 56.

```
private class HandlerMensajesConector extends Handler{
    public void handleMessage(Message msg){
        switch(msg.what){
        case ConstantesBiometrico.SOLICITAR_CONEXION_BT:
            if(dialogoDispositivosEntorno.isShowing())
                dialogoDispositivosEntorno.dismiss();
            Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableBtIntent, SOLICITAR_BLUETOOTH);
            break;
        case ConstantesBiometrico.BUSCANDO_DISPOSITIVOS:
            progresoBusqueda.setTitle("Buscando dispositivos...");
            progresoBusqueda.setMessage("Espere un momento...");
            progresoBusqueda.show();
            break;
```

Figura 56: Parte de código del Handler

En la figura se muestra sólo una parte del código. Lo que se hace es que, cada vez que el conector biométrico nos envía algún mensaje, se llama a "`handleMessage`". Como parámetro se pasa el mensaje, que tendrá una determinada acción. Se comprueba dicha acción con "`msg.what`", y, según cual sea, se procesa. El resto del código consiste en todas las posibles acciones que pueden llegar desde el conector biométrico.

Así, tras obtener el conector y seleccionar el handler que usaremos, procederemos a cargar la lista de dispositivos existentes en caso de haberlos. Esta lista mostrará un icono de dispositivo, junto al nombre y el estado de éste. Es por ello que hemos realizado un adaptador, ya que es de esta forma como conseguimos formatear una lista a nuestro gusto. Para ello se hace uso de la clase ListAdapterDispositivos.java.

A partir de aquí lo que se hace es esperar a eventos procedentes por parte del usuario. Existen varios listeners para atender a dichos eventos, que se producirán al pulsar los botones de Añadir dispositivo, Guardar o Cargar configuración.

A la hora de añadir, para la búsqueda de dispositivos se crea un diálogo con otra lista. Para darle formato, usamos otro adaptador ListAdapterBiometrico.java.

3.3.4.3. Clases de la Pantalla Medidas

Similar a la pantalla de dispositivos, la pantalla de medidas tiene la clase que construye el fragment MedidasFrag.java, además de una clase para formatear la lista de medidas cuando las pide el usuario.

Al entrar se mostrará la lista vacía, con los dos botones ya comentados, el referente a mostrar medidas y el referente a eliminar la base de datos. Estos botones tienen 2 listener a la espera de ser pulsados por el usuario. El código para realizar la petición de medidas se puede ver en la Figura 57.

```
public void mostrarMedidas(){
    SimpleDateFormat sdf = new SimpleDateFormat("dd/MMMM HH:mm");
    MedidasBBDDHelper bbdd = new MedidasBBDDHelper(contexto);
    List<MedidaDB> listaMedidas = bbdd.getMedidas();
    String[] listaMedidasArray = new String[listaMedidas.size()];
    MedidaDB medida;
    for(int i = 0; i < listaMedidas.size(); i++){
        medida = listaMedidas.get(i);
        String query = (i+1)+". Tipo: "+medida.getTipo()+"; Valor: "+medida.getValor()+medida.getUnidad()+"" +
                "; Localizacion: "+medida.getLocalizacion()+"; " +
                "Fecha: "+ sdf.format(new Date(medida.getFecha()));
        listaMedidasArray[i] = query;
    }
    listViewMedidas.setAdapter(new ListAdapterMedidas(getActivity(), R.layout.lista_entrada_temporal, listaMedidasArray){
        @Override
        public void onEntrada(Object entrada, View view) {
                TextView texto = (TextView)view.findViewById(R.id.textoMedida);
                texto.setText((String)entrada);
        }
    });
}
```

Figura 57: Código para mostrar medidas

Como vemos se hace uso del interfaz base de datos. Lo único que se realiza es instanciar el "gestor" de la base de datos, y a partir de éste obtener las medidas. Posteriormente se recorre la lista y gracias al adaptador ListAdapterMedidas.java se muestran en pantalla.

3.3.4.4. Clases de la Pantalla Gráfica

Para la última pantalla que vamos a especificar, la pantalla gráfica, se han definido tres clases, explicadas a continuación.

La primera de estas clases, GraficaFrag.java, sirve de contenedor para las otras dos. Es decir, es un fragment que a su vez tiene dos pestañas las cuales dan lugar a dos fragments diferentes.

El primero de ellos es el dedicado a la configuración de la gráfica, ConfigGraficaFrag.java. Su aspecto puede verse en la Figura 37. La implementación sigue el diagrama expuesto en la Figura 38. Existe además un listener para permitir seleccionar el intervalo de fechas entre las que se quieren representar unas determinadas medidas. Por otro lado están implementadas funciones para buscar medidas ambientales y medidas biométricas respectivamente, cuyos nombres son `buscarMedidasAmbientales` y `buscarMedidasBiometricas` respectivamente.

Por último, el método principal se encargará de, una vez buscadas todas las medidas, crear una instancia de Grafica.java del interfaz gráfico y cambiar a la pestaña de representación gráfica, RepresentacionGraficaFrag.java.

En esta última clase es donde se hace una llamada al método `mostrarGrafica` del interfaz gráfico, a la cual se pasa las instancias de Grafica.java antes creadas.

3.3.4.5. Clases auxiliares

Debido a la dificultad de determinar la localización del usuario mientras toma medidas, se ha decidido simular esta localización. Idealmente, debería cambiarse la posición según el usuario va moviéndose por el entorno en el que se encuentra la red de sensores inalámbricos instalada. Lo que hemos realizado es que cada un determinado periodo de tiempo, se seleccione una localización nueva de entre varias existentes en un *array* (una lista de elementos) con las localizaciones. Todo esto se lleva a cabo en la clase LocalizacionMovil.java.

Aunque la clase HandlerMedidasDispositivo.java la hemos considerado auxiliar, podría haberse codificado como una clase interna más en la clase dedicada a la pantalla de dispositivos. No obstante, la hemos codificado aparte para hacer el código más modular. Lo que realiza esta clase es obtener las medidas procedentes del dispositivo biométrico y almacenarlas en la base de datos local del Smartphone.

3.3.4.6. Driver

Para finalizar, tenemos la implementación del driver que se comunicará con el dispositivo biométrico. Existen dos clases que forman este driver, tal y como se describe a continuación.

La primera es un listener. Este listener se encarga de atender a eventos generados por el dispositivo biométrico, que, en nuestro caso, serán las medidas. Cuando exista una nueva medida, lo que hará este listener será enviarla por un handler que nosotros implementemos.

Por otro lado, tenemos el servicio, ServicioMedidas.java. Este es el que permanecerá en segundo plano atendiendo a todas las medidas procedentes del listener y enviándoselas al conector biométrico para que se encargue de procesarlas.

La codificación del handler no consideramos necesaria exponerla (se puede consultar en el código que se adjunta en formato electrónico a este documento). Análoga al resto de handlers, evaluará la acción del mensaje recibido (que en este caso será el tipo de medida) e informará al conector con los datos necesarios.

Por otro lado, en la Figura 58, se puede observar el código que principalmente se encarga de conectarse al dispositivo biométrico para comenzar la toma de medidas.

```
//Realizamos la conexión
btCliente = new BTClient(BluetoothAdapter.getDefaultAdapter(), direccionDispositivo);
listenerZephyr = new NewConnectedListener(handlerPulsometro, handlerPulsometro);
btCliente.addConnectedEventListener(listenerZephyr);
//Comprobamos que nos hemos conectado correctamente, e iniciamos la comunicación.
if(btCliente.IsConnected()){
    btCliente.start();
    //Informamos de que no podemos conectarnos al pulsómetro.
    avisarABiometrico(ConexionBiometricoHelper.BIOMETRICO_ACTIVADO);
}
else{//Si no está conectado, informamos a la actividad.
    avisarABiometrico(ConexionBiometricoHelper.BIOMETRICO_DESCONEXION_INVOLUNTARIA);
}
//Este servicio se ejecutará hasta que sea explícitamente parado
return START_STICKY;
```

Figura 58: Inicio de conexión con dispositivo biométrico

La instancia de BTClient será la encargada de llevar a cabo la comunicación. Se le pasan como parámetros el adaptador Bluetooth de Android y la dirección física del dispositivo. Posteriormente se le asocia el listener que escuchará a los eventos del pulsómetro, y a su vez este listener tendrá asociado el handler a través del cual nos comunicará dichos eventos. Por último, se realiza la conexión.

El método `avisarABiometrico` sirve para informar al conector biométrico de cualquier evento que ocurra respecto al dispositivo biométrico. Como comentamos anteriormente, el servicio implementa la interfaz ConexionBiometricoHelper, y, por tanto, nos obliga a escribir este método. La interfaz venía con una serie de constantes indicando los eventos que el conector biométrico espera recibir en algún momento. Dichos eventos los generaremos cuando consideremos necesario. En este trozo de código, por ejemplo, si la conexión se lleva a cabo correctamente, informamos al conector biométrico de que la conexión ha sido satisfactoria con el evento BIOMETRICO_ACTIVADO.

CAPÍTULO 4

PRUEBAS

Este capítulo lo dedicaremos a describir las diferentes pruebas realizadas sobre la aplicación Android desarrollada, así como sus resultados, para ver en funcionamiento todos los interfaces explicados en profundidad anteriormente. Consideramos interesante destacar que estas pruebas han sido realizadas tanto utilizando el emulador disponible como parte del SDK de Android como utilizando un terminal real (modelo Nexus 4) con este sistema operativo, en su versión 4.4.2, comunicándose mediante Bluetooth con un dispositivo biométrico real tal y como se ha indicado anteriormente en este documento.

4.1. Gestión de dispositivos: Agregar, modificar, o eliminar

Pasaremos en primer lugar a realizar las pruebas iniciales, consistentes en añadir un dispositivo, y posteriormente modificarlo e incluso eliminarlo.

Pulsamos en el botón Añadir dispositivo, y tras realizar la búsqueda, se nos muestran los dispositivos encontrados, como se puede apreciar en la Figura 59.

Figura 59: Búsqueda del Zephyr HxM

El nombre “HXM014292” es el nombre con el que el pulsómetro Zephyr viene de fábrica. Lo seleccionamos y pasamos al diálogo de añadir el dispositivo.

En la Figura 60 se puede observar una secuencia de 3 imágenes.

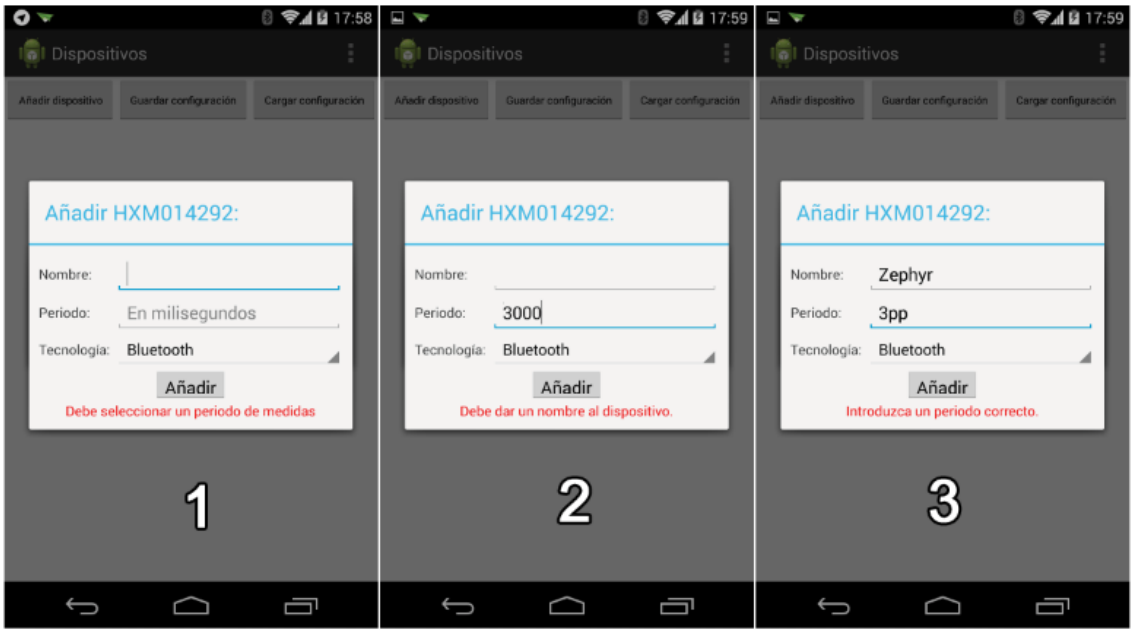

Figura 60: Pruebas para evitar añadir dispositivos incompletos

La primera de ellas, muestra un intento de añadir el dispositivo sin dotarle de un nombre amigable ni periodo, tras el cual se muestra un aviso de que debemos introducir un periodo.

En la segunda imagen, se ha introducido el periodo. No obstante, al no haberse dado un nombre, no se permite añadirlo tampoco.

Por último, se ha introducido nombre y periodo. Aun así, el periodo es erróneo, por lo que tampoco se permite añadir el dispositivo.

Se hace este hincapié en el nombre y en el periodo dado que estos parámetros forman parte de los principales (descritos en la especificación del interfaz biométrico) para el conector biométrico.

Una vez añadido, pasemos a ver cómo podríamos modificar parámetros. Aunque solamente se ha implementado la posibilidad de modificar el periodo, también podría darse la posibilidad de modificar el nombre.

En la Figura 61, hemos modificado el periodo, añadiendo una “k” al final. Al darle a atrás, se nos muestra un mensaje indicando que no se puede modificar el periodo debido a que no es correcto, y, por tanto, se deja intacto el que estaba anteriormente.

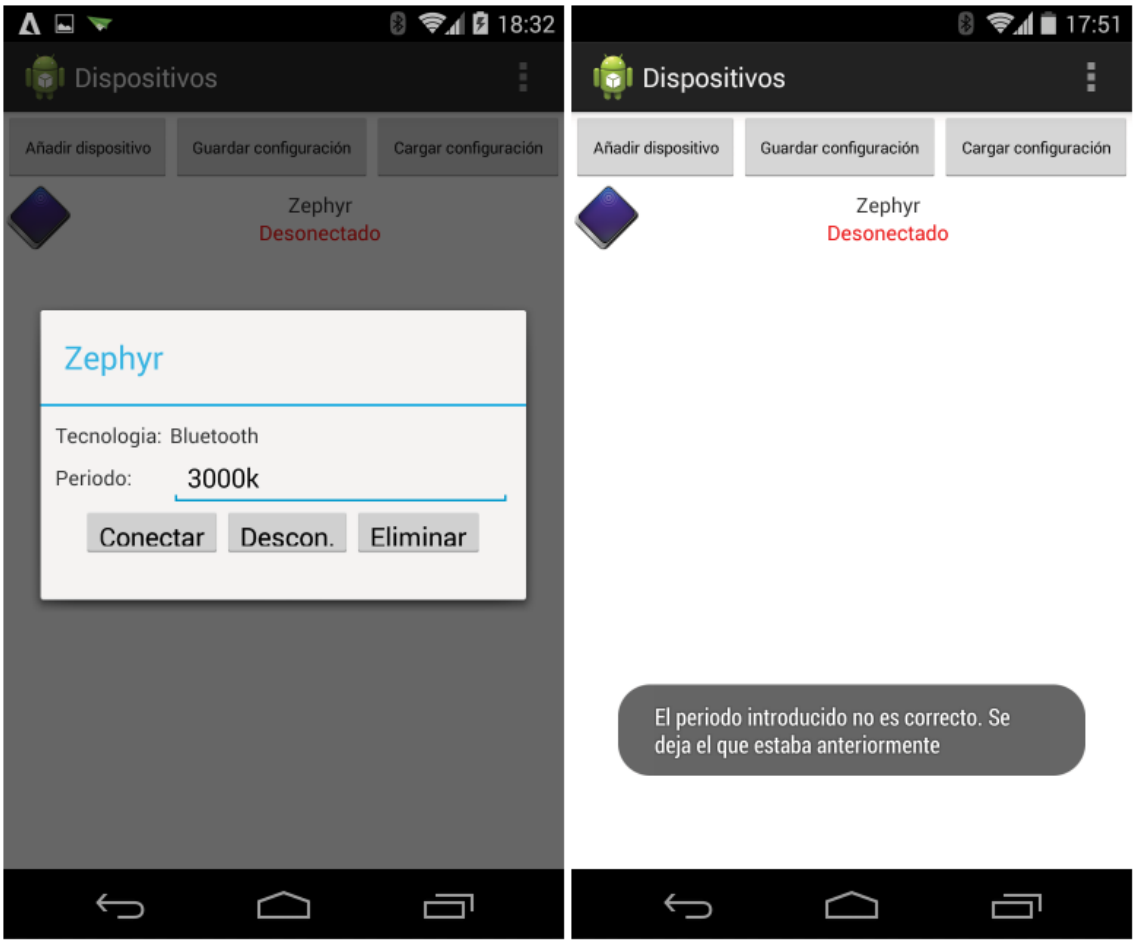

Figura 61: Edición del periodo y error

Por último, en la Figura 62 se puede observar el periodo modificado correctamente. Simplemente se nos pide que reconectemos el dispositivo en caso de que esté tomando medidas para que se apliquen los cambios correctamente.

Dado que el gestor no está programado para permitir modificaciones de dispositivos, lo que se hace es obtenerlo de la lista, modificar parámetros, y volver a añadirlo.

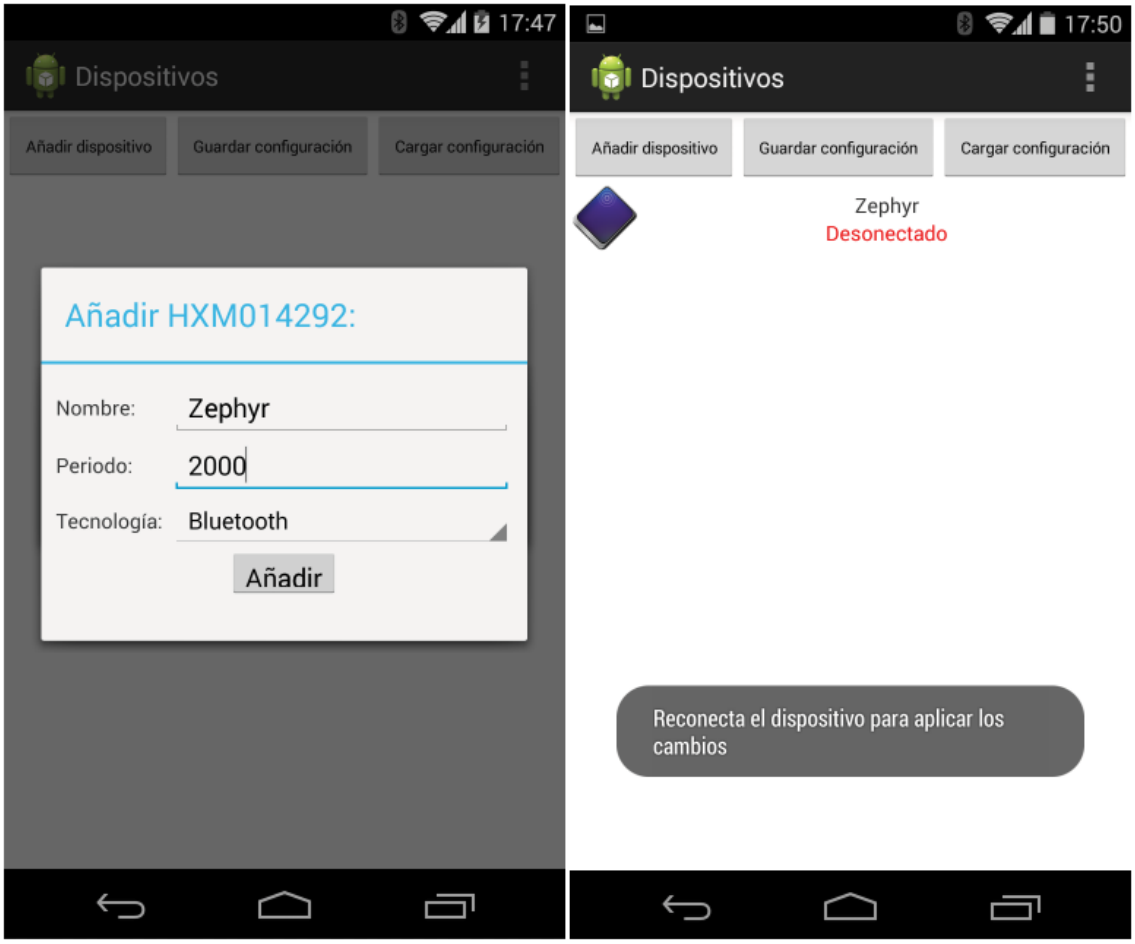

Figura 62: Edición del periodo satisfactoria

Por último, podemos eliminar el dispositivo. Esto simplemente elimina el dispositivo del gestor biométrico y, en nuestro caso, puesto que sólo existía un dispositivo en la lista, se muestra una lista vacía.

4.2. Toma de medidas

Una vez se añade el dispositivo, se puede empezar a tomar medidas de éste. Simplemente, en el diálogo que aparece al clicar en un dispositivo de la lista, procedemos a pulsar en "Conectar".

En la Figura 63 vemos el caso en el que, tras pulsar en *Conectar*, no se consigue realizar la conexión. Esto se debe a que el gestor trata de lanzar el servicio de medidas (o, como también lo hemos denominado a lo largo de este documento, el driver), quedando a la espera de ser informado por éste cuando se realice la conexión. Como vimos en el capítulo 3, se debe implementar una interfaz y tener en cuenta las constantes de ésta para que todo funcione correctamente. En este caso, el gestor ha recibido del driver la constante de la imposibilidad de conectar, por lo que el gestor informa al usuario del hecho.

Figura 63: Error de conexión

Cuando la conexión se realiza correctamente, vemos en la pantalla de la izquierda de la Figura 64 que el dispositivo aparece como “Conectado”. En este momento podemos navegar a otros menús de la aplicación o incluso ir fuera de ella, ya que la toma de medidas se realiza en segundo plano. En la imagen de la derecha podemos ver las medidas que se van tomando.

En el momento en que se desee parar la toma de medidas, simplemente habría que pulsar en el botón “Desconectar”, lo cual pararía el servicio.

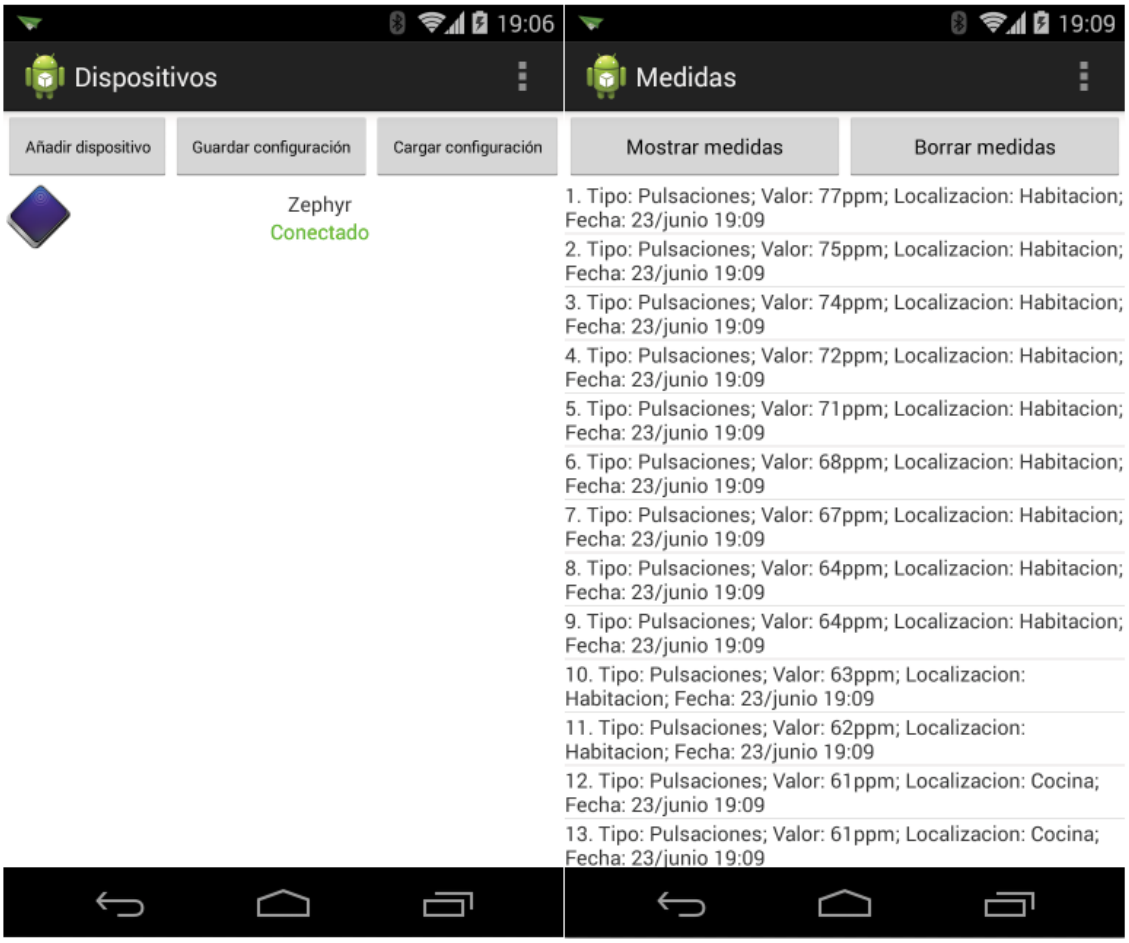

Figura 64: Lista de medidas almacenadas provenientes del Zephyr

4.3. Visualización gráfica

Para finalizar, veremos un ejemplo de gráfica generada a partir de las medidas tomadas tal y como se describe en el apartado anterior.

Como se comentó, las medidas ambientales se han simulado mediante un archivo XML que contiene una lista de medidas aleatorias, tomadas en localizaciones aleatorias.

En las opciones de configuración, los desplegables nos ofrecen los tipos de medida biométrica y ambiental disponibles, según las que soporta el conector biométrico y ambiental respectivamente.

Probemos primero a seleccionar una medida biométrica de la que no tenemos datos, para ver el comportamiento (ver Figura 65).

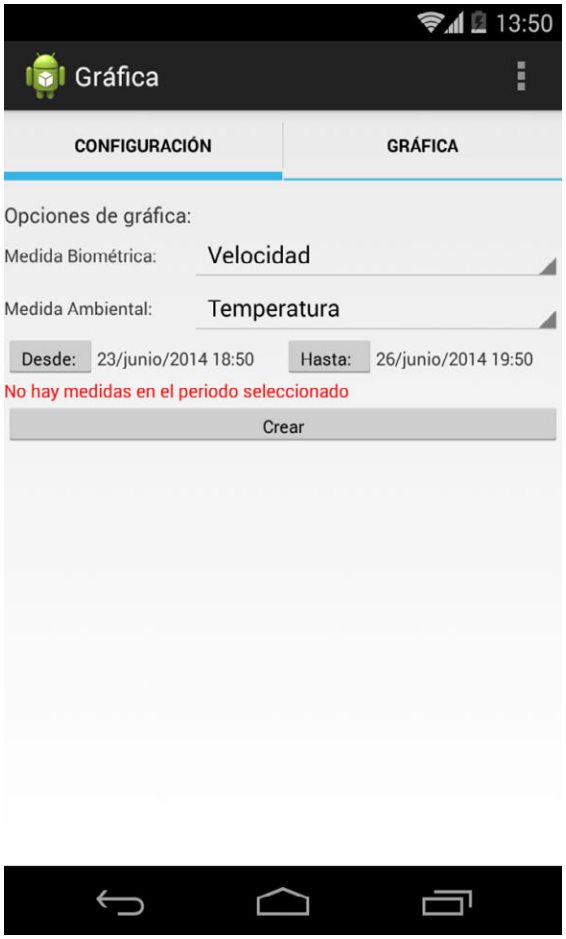

Figura 65: Error al no existir medidas seleccionadas en el periodo

Como vemos, se nos indica que no hay medidas en dicho periodo, por lo que no se mostrará ninguna gráfica, ya que el objetivo es relacionarlas con un determinado tipo de medida ambiental.

Seleccionamos ahora las pulsaciones como medida biométrica y humedad como medida ambiental, para un periodo temporal diferente. En este caso observamos que sí se muestra gráfica, aunque sin medidas de humedad ya que no existen. Esto es porque hemos considerado que se muestre la gráfica de medidas biométrica independientemente de que haya ambientales o no. En la Figura 66 se puede ver la pantalla representativa, con tamaño suficiente para apreciar los detalles.

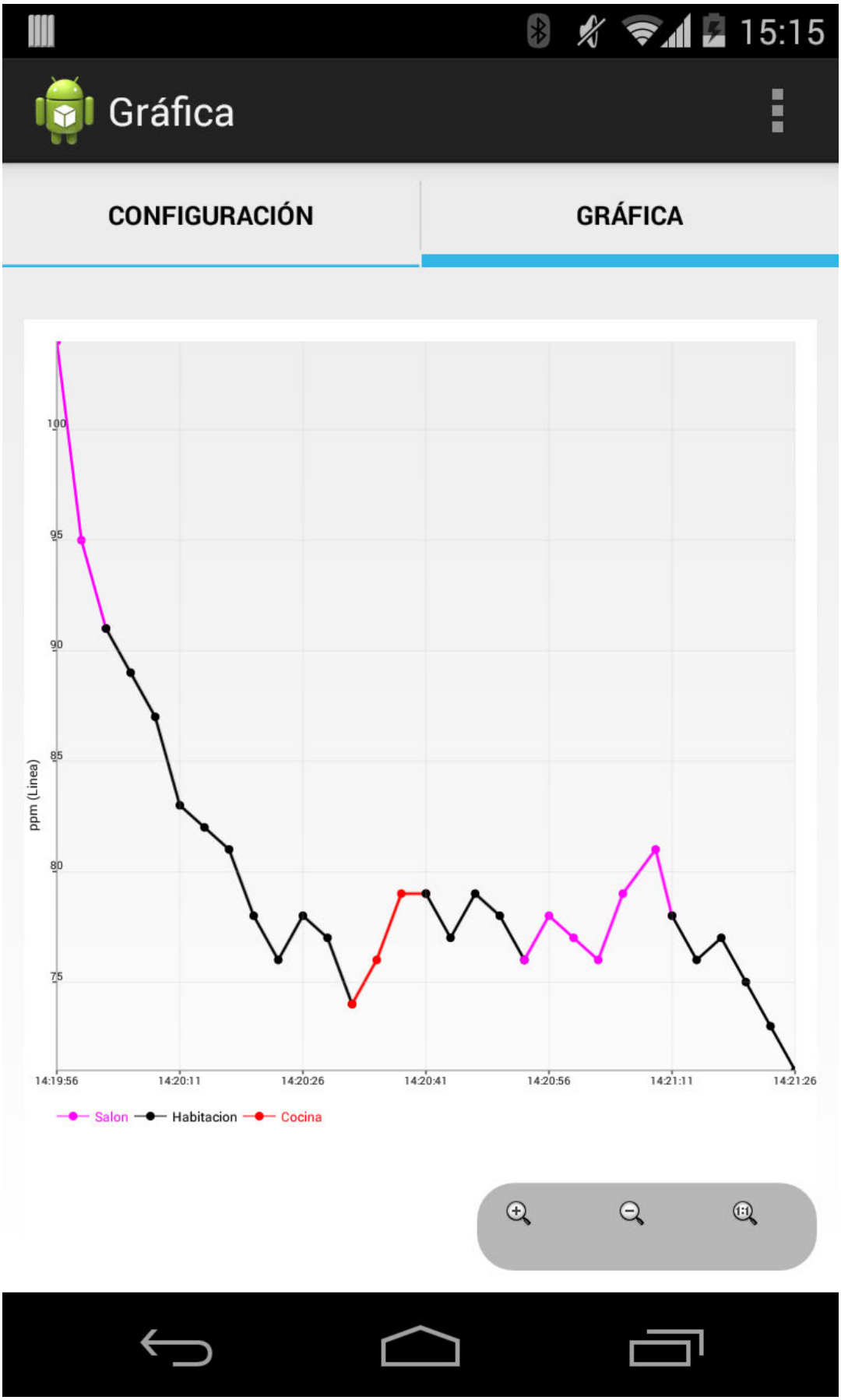

Figura 66: Gráfica sin medidas ambientales

Por último, probamos a seleccionar dos tipos de medida (una biométrica y otra ambiental) para un periodo para el que existen registros de ambas. En nuestro caso, escogeremos como medida biométrica el ritmo cardiaco y como medida ambiental la temperatura del entorno por el que se ha movido el usuario. La gráfica que se genera es la mostrada en la Figura 67.

Figura 67: Gráfica con medidas ambientales

Expliquemos con más detalle qué nos muestra la gráfica:

Se nos muestran dos ejes: Un eje con la unidad de las pulsaciones a la izquierda (pulsaciones por minuto – p.p.m.) indicando qué gráfica corresponde a las pulsaciones (en este caso, la línea), y otro eje con la unidad de temperatura a la derecha (ºC), indicando qué gráfica corresponde a la temperatura (en este caso, las barras). Podemos ver que existen tres colores. Cada localización está representada por un color. Dicho esto, vemos que, cuando el usuario se encontraba midiendo sus pulsaciones en el salón, la temperatura era de unos 24 grados. Es también interesante destacar que no es necesario que las muestras biométricas y ambientales estén tomadas en los mismos instantes de tiempo, de manera que no es necesaria una sincronización estricta entre el dispositivo biométrico y la red de sensores inalámbricos.

CAPÍTULO 5

CONCLUSIONES Y TRABAJOS FUTUROS

5.1. Conclusiones

Con el desarrollo de este proyecto buscábamos la creación de un sistema para *Smartphone* mediante el cual poder analizar mediciones provenientes de uno o más dispositivos biométricos y correlacionarlas con medidas ambientales tomadas por una red de sensores inalámbricos.

Los objetivos de esta correlación pueden ser diversos, pudiendo utilizarse las medidas con fines relacionados con la salud o con el deporte. No obstante, para aportar mayor flexibilidad y poder realizar un sistema con otros fines, se ha procedido a crear un conjunto de interfaces independientes entre sí.

Estos interfaces tienen funciones diferenciadas, destacando los interfaces biométrico y gráfico, y habiendo desarrollado los relacionados con el manejo de base de datos y la recuperación de medidas ambientales como asistentes para el usuario (entendido aquí como el programador de la aplicación para *Smartphone*).

Para la creación del interfaz biométrico se ha hecho uso de un dispositivo biométrico real, amoldando nuestro interfaz para lograr un alto nivel de abstracción y permitir así que todas las funciones que ofrece puedan utilizarse con cualquier otro dispositivo biométrico. El módulo de obtención de medidas biométricas que hemos realizado para ofrecer este interfaz implementa las funciones de gestión de los diferentes dispositivos que se agreguen, cumpliendo el que para nosotros era el objetivo más importante: lograr la abstracción que hemos comentado. Sin embargo, debido a las peculiaridades del dispositivo biométrico utilizado, la función de toma de medidas tiene que simular el periodo tras el cual se pide una medida al dispositivo, ya que el dispositivo no permite que se le pidan medidas, sino que las proporciona con una frecuencia determinada que no puede cambiarse.

Para el desarrollo del interfaz gráfico se ha buscado igualmente lograr la máxima simplicidad de uso, con un único objetivo en mente: que el usuario introduzca sus medidas y sean representadas en pantalla, diferenciando entre las biométricas y las ambientales, y diferenciando además dónde fueron tomadas todas esas medidas y en qué momento. Para ello el módulo de representación gráfica ofrece un único método al que simplemente hay que decirle las medidas que se desean representar. Además, está desarrollado para permitir mostrar simplemente un tipo de medidas en caso de no querer realizar correlación alguna, indicando claramente los instantes de las medidas y la localización del usuario al momento de tomarlas. Este módulo se ha implementado utilizando las funciones que ofrece la librería AChartEngine. Las mayores dificultades se han encontrado a la hora de realizar la correlación de medidas y distinguir las diferentes localizaciones. Como hemos comentado, en la gráfica se observa tanto el instante como la localización de las medidas. El instante queda claramente indicado en el eje temporal. Sin embargo, cuando se representan dos gráficas haciendo uso de la librería indicada, cada una hace uso de su eje de tiempos, no apareciendo ninguna relación entre ellas. Aunque hay una función de AChartEngine para permitir usar el

mismo eje para ambas gráficas, la falta de documentación de esta librería ha dificultado dar con dicho método, teniendo que acudir a consultar en numerosos foros. Para la localización, hemos procedido a colorear la gráfica en función de ésta. Debido a que AChartEngine no permite utilizar varios colores sino sólo uno por gráfica, la construcción de nuestra gráfica se basa en la unión de sub-gráficas que representan las medidas según las diferentes localizaciones, siendo parte del trabajo desarrollado en el proyecto ésta y otras tareas de adaptación de la funcionalidad ofrecida por la librería utilizada y las necesidades de la aplicación para *Smartphone*.

El interfaz de base de datos pretendía facilitar al programador que lo utilice el almacenamiento y recuperación de medidas biométricas. El módulo de almacenamiento y recuperación de medidas biométricas ofrece el interfaz consiguiendo esta facilidad al ocultar todos los detalles de cómo se debe hacer uso de una base de datos SQLite, que es la tecnología utilizada por Android.

Por último, el interfaz ambiental buscaba una función similar a la del interfaz base de datos: la recuperación de medidas, aunque, en este caso, ambientales. Es por ello que el interfaz ofrece únicamente al programador la posibilidad de recuperar una lista de medidas ambientales según unas fechas determinadas (inicio y final) y tipo de medida. El módulo de obtención de medidas ambientales desarrollado simula un servidor que almacena dichas medidas, haciendo realmente uso de un fichero XML que contiene todas las medidas que pudieran haber sido hipotéticamente tomadas por una red de sensores inalámbricos.

Todo el sistema se ha realizado para ser implementado en una plataforma Android, por lo que ha sido desarrollado en su totalidad en Java, haciendo uso de las librerías que ofrece el SDK de ese sistema operativo. Se ha escrito un total de aproximadamente 3000 líneas de código Java entre la aplicación Android y los módulos para poder ofrecer todas las funcionalidades planteadas en un principio.

El desarrollo de este proyecto ha sido de gran utilidad para conocer con claridad cómo deben desarrollarse unos interfaces y cómo hacer uso de ellos. A la hora de diseñar los interfaces hay que tener muy claro qué funciones son las que ofrecerán estas interfaces y tener una idea de cómo podrían implementarse para que no surjan demasiadas dificultades a la hora de hacerlo. En el proceso de diseño se ha tenido en cuenta en todo momento que deben aportar una flexibilidad suficiente para que puedan ampliarse sus funciones en un futuro sin afectar a lo ya existente. Por último, se ha cumplido la independencia total entre todos los interfaces diseñados, pudiendo utilizar simplemente uno o todos ellos según se requiera.

Además, este proyecto ha ayudado también a aprender y asentar conocimientos en el desarrollo de una aplicación Android: qué partes forman la aplicación, el ciclo de vida de ésta, cómo se comunican las distintas partes que la componen e incluso cómo hacer uso de la tecnología que ofrece el teléfono para comunicarnos con dispositivos externos. De manera más general, también se han adquirido nociones de las particularidades que tiene el diseño y desarrollo de sistemas para plataformas móviles, complementarias a las

necesarias para el desarrollo de sistemas pensados para ejecutarse en equipos de tipo ordenador personal convencional.

Dicho todo esto, podemos dar por cumplidos no sólo todos los objetivos técnicos planteados al principio de este documento sino también los objetivos de formación y aplicación de diversos conocimientos que son inherentes a un Proyecto Fin de Grado.

5.2. Trabajos futuros

Enumeraremos en este apartado diferentes propuestas para futuras líneas de trabajo a seguir a partir de este trabajo.

5.2.1. Mejoras en los interfaces y aplicación Android

- Soporte de más dispositivos biométricos: Sería interesante, aparte de dar la posibilidad al usuario de desarrollar los servicios de toma de medidas como él quiera, ofrecer soporte para distintos dispositivos existentes en el mercado, de forma que se simplifique aún más el uso del interfaz.

- Posibilitar la eliminación de medidas específicas: Actualmente sólo se permite borrar todas las medidas guardadas, dado que se elimina la base de datos completa.

- Dotar de personalización a la gráfica: Por ejemplo, se podrían añadir controles para permitir al usuario mostrar u ocultar medidas según localización, o elegir colores libremente.

- Gráfica en tiempo real: Esta nueva funcionalidad consistiría en realizar una gráfica en la que aparezcan las medidas de manera dinámica según van siendo tomadas.

- Mejora de la experiencia de usuario: La aplicación ha sido realizada con el objetivo de mostrar el funcionamiento de los interfaces y realizar el sistema para correlar medidas, requisitos que han sido cubiertos. No obstante, la experiencia del usuario a la hora de utilizar la aplicación no ha recibido la mayor prioridad, por lo que mejorar este aspecto teniendo en cuenta criterios de diseño de interfaces de usuario constituye otro posible campo de mejora.

CAPÍTULO 6

BIBLIOGRAFÍA

[1] Waltenegus Dargie and Christian Poellabauer, *Fundamentals of Wireless Sensor Netowkrs. Theory and Practice*.: Wiley, 2010. Consultado en Mayo de 2014.

[2] Direct Industry. Nodo inalámbrico V-Link. [Online]. Consultado en Mayo de 2014. http://www.directindustry.com/prod/microstrain/wireless-sensor-networks-wsn-27212-347848.html

[3] I.F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E Cayirci, "A survey on sensor networks," *IEEE Communications Magazine*, vol. 40, no. 8, pp. 102,114, 2002. Consultado en Mayo de 2014.

[4] M.A. Matin and M.M. Islam, "Overview of Wireless Sensor Network," in *Wireless Sensor Networks - Technology and Protocols*.: Intech, 2012, ch. 1. Consultado en Mayo de 2014.

[5] Sitio web del sistema operativo iOS. [Online]. Consultado en Mayo de 2014. http://www.apple.com/es/ios/

[6] Android. [Online]. Consultado en Mayo de 2014. http://www.android.com

[7] Sitio web de Open Handset Alliance. Consultado en Mayo de 2014. [Online]. http://www.openhandsetalliance.com/

[8] Sitio web de Windows Phone. [Online]. Consultado en Mayo de 2014. http://www.windowsphone.com/es-ES

[9] Oliver Frommel. (2013, Febrero) Developing an App for iOS, Android and Windows Phone - a Comparative Study. [Online]. Consultado en Mayo de 2014. http://www.zetalab.de/blog/developing-an-app-for-ios-android-and-windows-phone-a-comparative-study/

[10] Matías S. Zavia. (2014, Abril) Cuota de mercado de Android en España a Marzo 2014. [Online]. Consultado en Mayo de 2014. http://www.xatakamovil.com/mercado/ios-y-windows-phone-le-quitan-cuota-de-mercado-a-android-en-espana-con-mucha-timidez

[11] SomosLibres. (2014, Marzo) Android configura renovada arquitectura del kernel de Linux. [Online]. Consultado en Mayo de 2014. http://www.somoslibres.org/modules.php?name=News&file=article&sid=5362

[12] Basterra - Bertea - Borello - Castillo - Venturi. (2012) Historia de Android. [Online]. Consultado en Mayo de 2014. http://androidos.readthedocs.org/en/latest/data/historia/

[13] Quentyn Kennemer. (2013, Noviembre) Nova Launcher beta update brings transparent notification bar and nav buttons on Android 4.4+. [Online]. http://phandroid.com/2013/11/11/nova-launcher-beta-update/

[14] Características de Android. [Online]. http://es.wikipedia.org/wiki/Android#Caracter.C3.ADsticas

[15] Sitio web de Google Glass. [Online]. Consultado en Junio de 2014. https://www.google.com/glass/start/

[16] Times Internet Limited. (2014, Febrero) Samsung unveils wearable devices: Gear Fit & Gear 2. [Online]. Consultado en Junio de 2014. http://economictimes.indiatimes.com/samsung-unveils-wearable-devices/slideshow/30981896.cms

[17] Gabriela Gottau. (2014, Enero) Razer Nabu, una pulsera más para cuantificar la actividad del día a día. [Online]. Consultado en Junio de 2014. http://www.vitonica.com/equipamiento/razer-nabu-una-pulsera-mas-para-cuantificar-la-actividad-del-dia-a-dia

[18] Polar Electro. Producto Polar FT7: Pulsómetro de entrenamiento. Consultado en Junio de 2014. [Online]. http://www.polar.com/es/productos/get_active/fitness_crosstraining/FT7

[19] Sitio web de Zephyr BT HxM. [Online]. Consultado en Junio de 2014. http://zephyranywhere.com/products/hxm-bluetooth-heart-rate-monitor/

[20] BlackBerry Empire Contributor. (2014, Abril) Zephyr HxM – Delivers Heart Rate, Speed & Distance to BlackBerry Smartphones. [Online]. Consultado en Junio de 2014. http://blackberryempire.com/zephyr-hxm-delivers-heart-rate-speed-distance-to-blackberry-smartphones/

[21] Apps para el Zephyr BT HxM. [Online]. Consultado en Junio de 2014. http://zephyranywhere.com/apps/

[22] Unified Modified Language. [Online]. Consultado en Junio de 2014. http://www.uml.org/

[23] Sitio web de Eclipse. [Online]. Consultado en Junio de 2014. http://www.eclipse.org/

[24] Sitio web del SDK de Android. Consultado en Junio de 2014. [Online]. http://developer.android.com/sdk/index.html

[25] Sitio web de AChartEngine. Consultado en Junio de 2014. [Online]. http://www.achartengine.org/

[26] Sitio web de StackOverflow. Consultado en Junio de 2014. [Online]. http://stackoverflow.com/

[27] Judith Bishop, "Creational Patterns: Prototype, Factory Method, and Singleton," in *C# 3.0 Design Patterns: Use the Power of C# 3.0 to Solve Real-World Problems*.: O'Reilly Media, 2007. [Online]. Consultado en Junio de 2014. http://msdn.microsoft.com/en-us/library/orm-9780596527730-01-05.aspx

[28] Explicación de POSIX. [Online]. Consultado en Junio de 2014. http://es.wikipedia.org/wiki/POSIX

[29] Explicación de los Fragments de Android Developer. Consultado en Junio de 2014. [Online]. http://developer.android.com/guide/components/fragments.html

[30] Creación de un Navigation Drawer. Consultado en Junio de 2014. [Online]. http://developer.android.com/training/implementing-navigation/nav-drawer.html

ANEXO

DOCUMENTACIÓN DE LOS MÓDULOS DESARROLLADOS

Documentación

6/28/14 7:14 PM

Package Summary		Page
com.example.pfg.interfazAmbiental		102
com.example.pfg.interfazBBDD		109
com.example.pfg.interfazBiometrico		117
com.example.pfg.interfazGrafico		145

Package com.example.pfg.interfazAmbiental

Class Summary		Page
GestorAmbientales		103
MedidaAmbiental	Modelo de medida utilizado por el interfaz ambiental	106

Class GestorAmbientales

com.example.pfg.interfazAmbiental

```
java.lang.Object
  └ com.example.pfg.interfazAmbiental.GestorAmbientales
```

```
public class GestorAmbientales
extends Object
```

Field Summary		Page
static String	TIPO_MEDIDA_TEMPERATURA	104

Constructor Summary	Page
GestorAmbientales(Context c)	105

Method Summary		Page
ArrayList<MedidaAmbiental>	getMedidasAmbientales() Obtiene todas las medidas ambientales existentes	105
ArrayList<MedidaAmbiental>	getMedidasAmbientalesPorFecha(Date fechaInicial, Date fechaFin) Obtiene una serie de medidas ambientales comprendidas entre dos fechas	105
ArrayList<MedidaAmbiental>	getMedidasAmbientalesPorFechaYTipo(long fechaInicial, long fechaFin, String tipo, String localizacion) Obtiene una lista de medidas ambientales comprendidas entre una fecha, de un determinado tipo y según una localización determinada.	106
String[]	getMedidasDisponibles() Obtiene los tipos de medidas ambientales disponibles	105

Field Detail

TIPO_MEDIDA_TEMPERATURA

```
public static final String TIPO_MEDIDA_TEMPERATURA
```

Constructor Detail

GestorAmbientales

```
public GestorAmbientales(Context c)
```

Method Detail

getMedidasAmbientales

```
public ArrayList<MedidaAmbiental> getMedidasAmbientales()
```

Obtiene todas las medidas ambientales existentes

---

getMedidasDisponibles

```
public String[] getMedidasDisponibles()
```

Obtiene los tipos de medidas ambientales disponibles

Returns:

La lista de tipos de medidas

---

getMedidasAmbientalesPorFecha

```
public ArrayList<MedidaAmbiental> getMedidasAmbientalesPorFecha(Date fechaInic
ial,
                                                                Date fechaFin)
```

Obtiene una serie de medidas ambientales comprendidas entre dos fechas

Parameters:

`fechaInicial` - - La fecha de inicio

`fechaFin` - - La fecha final

Returns:

La lista de medidas existentes entre esas fechas, o una lista vacía si no hay.

---

getMedidasAmbientalesPorFechaYTipo

```
public ArrayList<MedidaAmbiental> getMedidasAmbientalesPorFechaYTipo(long fech
aInicial,
                                                                     long fech
aFin,
                                                                     String ti
po,
                                                                     String lo
calizacion)
```

Obtiene una lista de medidas ambientales comprendidas entre una fecha, de un determinado tipo y según una localización determinada.

Parameters:

`fechaInicial` - - La fecha de inicio

`fechaFin` - - La fecha final

`tipo` - - El tipo de medida

`localizacion` - - La localización de la medida

Returns:

La lista con las medidas o una lista vacía en caso de no haber.

Class MedidaAmbiental

com.example.pfg.interfazAmbiental

```
java.lang.Object
```

└ **com.example.pfg.interfazAmbiental.MedidaAmbiental**

All Implemented Interfaces:

Serializable

```
public class MedidaAmbiental

extends Object

implements Serializable
```

Modelo de medida utilizado por el interfaz ambiental

Constructor Summary	*Page*
`MedidaAmbiental()`	*107*
`MedidaAmbiental(String tipo, String valor, String unidad, String localizacion, long fecha)`	*107*

Method Summary		Page
long	**getFecha**()	108
String	**getLocalizacion**()	108
String	**getTipo**()	107
String	**getUnidad**()	108
String	**getValor**()	107
void	**setFecha**(long fecha)	108
void	**setLocalizacion**(String localizacion)	108
void	**setTipo**(String tipo)	107
void	**setUnidad**(String unidad)	108
void	**setValor**(String valor)	107

Constructor Detail

MedidaAmbiental

```
public MedidaAmbiental(String tipo,
                       String valor,
                       String unidad,
                       String localizacion,
                       long fecha)
```

MedidaAmbiental

```
public MedidaAmbiental()
```

Method Detail

getTipo

```
public String getTipo()
```

---

setTipo

```
public void setTipo(String tipo)
```

---

getValor

```
public String getValor()
```

---

setValor

```
public void setValor(String valor)
```

---

getUnidad

```
public String getUnidad()
```

---

setUnidad

```
public void setUnidad(String unidad)
```

---

getLocalizacion

```
public String getLocalizacion()
```

---

setLocalizacion

```
public void setLocalizacion(String localizacion)
```

---

getFecha

```
public long getFecha()
```

---

setFecha

```
public void setFecha(long fecha)
```

Package com.example.pfg.interfazBBDD

Class Summary		Page
MedidaDB		109
MedidasBBDDHelper	Gestor de base de datos de medidas que abstrae al usuario de operaciones SQL.	113

Class MedidaDB

com.example.pfg.interfazBBDD

```
java.lang.Object
  └ com.example.pfg.interfazBBDD.MedidaDB
```

All Implemented Interfaces:
Serializable

```
public class MedidaDB
extends Object
implements Serializable
```

Constructor Summary	Page
MedidaDB()	111
MedidaDB(String tipo, String valor, String unidad, String localizacion, long fecha)	110

Method Summary		Page
long	getFecha()	111
String	getLocalizacion()	111
String	getTipo()	111
String	getUnidad()	111
String	getValor()	111
void	setFecha(long fecha)	111
void	setLocalizacion(String localizacion)	112
void	setTipo(String tipo)	111
void	setUnidad(String unidad)	111
void	setValor(String valor)	111

Constructor Detail

MedidaDB

```
public MedidaDB(String tipo,
                String valor,
                String unidad,
                String localizacion,
                long fecha)
```

MedidaDB

```
public MedidaDB()
```

Method Detail

getTipo

```
public String getTipo()
```

---

setTipo

```
public void setTipo(String tipo)
```

---

getValor

```
public String getValor()
```

---

setValor

```
public void setValor(String valor)
```

---

getUnidad

```
public String getUnidad()
```

---

setUnidad

```
public void setUnidad(String unidad)
```

---

getFecha

```
public long getFecha()
```

---

setFecha

```
public void setFecha(long fecha)
```

---

getLocalizacion

```
public String getLocalizacion()
```

setLocalizacion

```
public void setLocalizacion(String localizacion)
```

Class MedidasBBDDHelper

com.example.pfg.interfazBBDD

```
java.lang.Object
  └ SQLiteOpenHelper
      └ com.example.pfg.interfazBBDD.MedidasBBDDHelper
```

```
public class MedidasBBDDHelper
extends SQLiteOpenHelper
```

Gestor de base de datos de medidas que abstrae al usuario de operaciones SQL.

Field Summary		Page
static String	DB_NAME	114
static int	DB_VERSION	114
static String	KEY_FECHA	115
static String	KEY_LOCALIZACION	114
static String	KEY_TIPO	114
static String	KEY_UNIDAD	114
static String	KEY_VALOR	114
static String	TABLE_MEDIDAS	114

Constructor Summary	Page
MedidasBBDDHelper(Context context)	115

Method Summary		Page
void	addMedida(MedidaDB medida) Añadirá una nueva medida a la base de datos.	115
void	eliminarBBDD() Elimina la base de datos del terminal móvil.	116
List<MedidaDB>	getMedidas() Obtiene todas las medidas existentes en la base de datos	116

List<MedidaDB>	**getMedidas**(String fechaInicio, String fechaFin, String tipo) Obtiene una serie de medidas comprendidas entre dos fechas.	*115*
void	**onCreate**(SQLiteDatabase db) Creará la base de datos cuando sea necesario.	*115*
void	**onUpgrade**(SQLiteDatabase db, int oldVersion, int newVersion) Actualizará la base de datos cuando sea necesario modificar la estructura de la actual.	*115*

Field Detail

DB_NAME

```
public static final String DB_NAME
```

DB_VERSION

```
public static final int DB_VERSION
```

TABLE_MEDIDAS

```
public static final String TABLE_MEDIDAS
```

KEY_TIPO

```
public static final String KEY_TIPO
```

KEY_VALOR

```
public static final String KEY_VALOR
```

KEY_UNIDAD

```
public static final String KEY_UNIDAD
```

KEY_LOCALIZACION

```
public static final String KEY_LOCALIZACION
```

KEY_FECHA

```
public static final String KEY_FECHA
```

Constructor Detail

MedidasBBDDHelper

```
public MedidasBBDDHelper(Context context)
```

Method Detail

onCreate

```
public void onCreate(SQLiteDatabase db)
```

Creará la base de datos cuando sea necesario.

---

onUpgrade

```
public void onUpgrade(SQLiteDatabase db,
                      int oldVersion,
                      int newVersion)
```

Actualizará la base de datos cuando sea necesario modificar la estructura de la actual.

---

addMedida

```
public void addMedida(MedidaDB medida)
```

Añadirá una nueva medida a la base de datos.

Parameters:

`medida` - - La medida a añadir

---

getMedidas

```
public List<MedidaDB> getMedidas(String fechaInicio,
                                 String fechaFin,
                                 String tipo)
```

Obtiene una serie de medidas comprendidas entre dos fechas. Las fechas deben estar en formato UNIX.

Parameters:

`fechaInicio` - - La fecha inicial

`fechaFin` - - La fecha final

`tipo` - - El tipo de medida

Returns:

Lista con las medidas encontradas con las condiciones indicadas.

---

eliminarBBDD

```
public void eliminarBBDD()
```

Elimina la base de datos del terminal móvil.

---

getMedidas

```
public List<MedidaDB> getMedidas()
```

Obtiene todas las medidas existentes en la base de datos

Returns:

Lista con todas las medidas que se encuentren en la base de datos.

Package com.example.pfg.interfazBiometrico

Interface Summary		Page
ConexionBiometricoHelper	Interfaz utilizada para enviar avisos al conector biométrico desde el servicio de medidas que implemente el usuario.	127

Class Summary		Page
BiometricoBT	Clase para realizar conexiones y configuraciones relativas a dispositivos Bluetooth.	117
ConexionBiometrico	Asistente para llevar a cabo la conexión con un dispositivo biométrico que use cualquiera de las tecnologías soportadas por el teléfono móvil.	121
ConstantesBiometrico	Define constantes referentes a todo lo relacionado con la conexión biométrica.	130
Dispositivo	Modelo de dispositivo del que hace uso el paquete biométrico.	133
Medida	Modelo de medida utilizado por el paquete biométrico.	140
ParametrosServicio	La clase ParametrosServicio ofrece la posibilidad de pasar al servicio de medidas creado una serie de parámetros que requiera el usuario porque así lo necesite.	143

Exception Summary		Page
ErrorBiometricoException		139

Class BiometricoBT

com.example.pfg.interfazBiometrico

```
java.lang.Object
  └ com.example.pfg.interfazBiometrico.BiometricoBT
```

```
public class BiometricoBT
extends Object
```

Clase para realizar conexiones y configuraciones relativas a dispositivos Bluetooth.

Field Summary		Page
static int	BT_MEDIDA	119
static int	BT_SOLICITAR_CONEXION	119
static int	BUSCANDO_DISPOSITIVOS	119
static int	BUSQUEDA_FINALIZADA	119
static int	INICIANDO_CONEXION	119
static int	NO_SOPORTA_BT	119
static int	NUEVO_DISPOSITIVO	119
static String	OBTENER_DIRECCION	119
static String	OBTENER_DISPOSITIVO	119

Constructor Summary	Page
BiometricoBT(Handler handlerMensajesHaciaConexionBiometrico, Context context)	120

Method Summary		Page
void	mostrarListaDispositivosBTExistentes() Buscará los dispositivos Bluetooth alcanzables por el dispositivo móvil.	120
String	obtenerDireccion(String nombreDispositivo) Obtiene la dirección física de un dispositivo	120

Field Detail

INICIANDO_CONEXION

```
public static final int INICIANDO_CONEXION
```

NO_SOPORTA_BT

```
public static final int NO_SOPORTA_BT
```

BT_SOLICITAR_CONEXION

```
public static final int BT_SOLICITAR_CONEXION
```

BT_MEDIDA

```
public static final int BT_MEDIDA
```

BUSCANDO_DISPOSITIVOS

```
public static final int BUSCANDO_DISPOSITIVOS
```

BUSQUEDA_FINALIZADA

```
public static final int BUSQUEDA_FINALIZADA
```

NUEVO_DISPOSITIVO

```
public static final int NUEVO_DISPOSITIVO
```

OBTENER_DISPOSITIVO

```
public static final String OBTENER_DISPOSITIVO
```

OBTENER_DIRECCION

```
public static final String OBTENER_DIRECCION
```

Constructor Detail

BiometricoBT

```
public BiometricoBT(Handler handlerMensajesHaciaConexionBiometrico,
                    Context context)
```

Method Detail

mostrarListaDispositivosBTExistentes

```
public void mostrarListaDispositivosBTExistentes()
```

Buscará los dispositivos Bluetooth alcanzables por el dispositivo móvil. Cada vez que encuentre uno nuevo, se devolverá la acción BiometricoBT.NUEVO_DISPOSITIVO, y en el mensaje irá un bundle con el nombre del dispositivo, cuyo identificador es BiometricoBT.OBTENER_DISPOSITIVO.

---

obtenerDireccion

```
public String obtenerDireccion(String nombreDispositivo)
```

Obtiene la dirección física de un dispositivo

Parameters:

`nombreDispositivo` - - El nombre del dispositivo del que se quiere la dirección.

Returns:

La dirección del dispositivo

Class ConexionBiometrico

com.example.pfg.interfazBiometrico

java.lang.Object

└ **com.example.pfg.interfazBiometrico.ConexionBiometrico**

---

public class **ConexionBiometrico**

extends Object

Asistente para llevar a cabo la conexión con un dispositivo biométrico que use cualquiera de las tecnologías soportadas por el teléfono móvil. Mantiene una lista de dispositivos a los cuales se tratará de conectar.

Field Summary		Page
static String	**MEDIDA_TIPO_HEARTRATE**	122
static String	**MEDIDA_TIPO_VELOCIDAD**	122

Method Summary		Page
void	**addDispositivoConectable**(Dispositivo dispositivo) Añade un dispositivo biométrico al conector.	125
ArrayList<Dispositivo>	**cargarConfiguracionBiometrico**(String nombreFichero) Devuelve una lista de dispositivos biométricos a partir de un fichero de configuración.	123
void	**eliminarDispositivo**(String nombreAmigable) Elimina un dispositivo del conector biométrico.	123
static ConexionBiometrico	**getConectorBiometrico**(Context c) Obtiene una instancia del conector biométrico.	122
Dispositivo	**getDispositivo**(String nombreAmigable) Devuelve un dispositivo de la lista de dispositivos que almacena el conector biométrico.	125
ArrayList<Dispositivo>	**getListaDispositivos**() Obtiene la lista de dispositivos añadidos al conector biométrico.	125
String[]	**getMedidasDisponibles**()	124
int[]	**getTecnologiasDisponibles**()	124
String[]	**getTecnologiasString**()	124

boolean	**guardarConfiguracionBiometrico**(String nombreFichero) Guarda la configuración actual de todos los dispositivos que se encuentren en el conector biométrico.	*123*
void	**iniciarMedidas**(String nombreAmigable) Se encargará de conectar a un dispositivo biométrico para comenzar la toma de medidas.	*126*
void	**mostrarDispositivos**(int tecnologia) Iniciará una búsqueda de dispositivos.	*125*
String	**obtenerDireccion**(String nombreDispositivo, int tecnologia) Obtiene la dirección física de un determinado dispositivo.	*123*
void	**pararMedidas**(String nombreAmigable) Desconectará un dispositivo y parará la toma de medidas.	*126*
void	**setHandlerHaciaCaller**(Handler handlerHaciaCaller) Establece el hanlder a través del cual se comunicará el conector biométrico con el usuario.	*124*

Field Detail

MEDIDA_TIPO_HEARTRATE

```
public static String MEDIDA_TIPO_HEARTRATE
```

MEDIDA_TIPO_VELOCIDAD

```
public static String MEDIDA_TIPO_VELOCIDAD
```

Method Detail

getConectorBiometrico

```
public static ConexionBiometrico getConectorBiometrico(Context c)
```

Obtiene una instancia del conector biométrico.

Parameters:

c - - Contexto de la aplicación.

Returns:

Una instancia de conector biometrco.

---

guardarConfiguracionBiometrico

```
public boolean guardarConfiguracionBiometrico(String nombreFichero)
```

Guarda la configuración actual de todos los dispositivos que se encuentren en el conector biométrico. No se guardarán los manejadores asociados a dichos dispositivos.

Parameters:

`nombreFichero` - - Nombre del fichero en el que se guardará la configuración.

Returns:

true si se ha guardado correctamente, false en caso contrario.

---

cargarConfiguracionBiometrico

```
public ArrayList<Dispositivo> cargarConfiguracionBiometrico(String nombreFiche
ro)
```

Devuelve una lista de dispositivos biométricos a partir de un fichero de configuración. Se deben asociar los Handlers correspondientes.

Parameters:

`nombreFichero` - - Nombre del fichero con la configuración a cargar.

Returns:

La lista de dispositivos con sus respectivas configuraciones.

---

eliminarDispositivo

```
public void eliminarDispositivo(String nombreAmigable)
```

Elimina un dispositivo del conector biométrico.

Parameters:

`nombreAmigable` - - Nombre del dispositivo a elimiar.

---

obtenerDireccion

```
public String obtenerDireccion(String nombreDispositivo,
                               int tecnologia)
```

Obtiene la dirección física de un determinado dispositivo.

Parameters:

`nombreDispositivo` - - El nombre del dispositivo.

`tecnologia` - - La tecnología que utiliza el dispositivo.

Returns:

La dirección del dispositivo solicitado.

---

getTecnologiasDisponibles

`public int[] getTecnologiasDisponibles()`

Returns:

Obtiene la lista de tecnologías disponibles a usar en el conector biométrico, en tipo int

---

getTecnologiasString

`public String[] getTecnologiasString()`

Returns:

Obtiene la lista de tecnologías disponibles a usar en el conector biométrico, en tipo String

---

getMedidasDisponibles

`public String[] getMedidasDisponibles()`

Returns:

Obtiene una lista de tipos de medidas que soporta el conector biométrico.

---

setHandlerHaciaCaller

`public void setHandlerHaciaCaller(Handler handlerHaciaCaller)`

Establece el hanlder a través del cual se comunicará el conector biométrico con el usuario.

Parameters:

`handlerHaciaCaller` - - la referencia al handler

---

addDispositivoConectable

```
public void addDispositivoConectable(Dispositivo dispositivo)
                              throws ErrorBiometricoException
```

Añade un dispositivo biométrico al conector.

Parameters:

`dispositivo` - - El dispositivo a añadir

Throws:

`ErrorBiometricoException` - - Si el dispositivo ya existe.

---

getDispositivo

```
public Dispositivo getDispositivo(String nombreAmigable)
```

Devuelve un dispositivo de la lista de dispositivos que almacena el conector biométrico.

Parameters:

`nombreAmigable` - - Nombre del dispositivo a devolver

Returns:

El dispositivo o null en caso de no existir.

---

getListaDispositivos

```
public ArrayList<Dispositivo> getListaDispositivos()
```

Obtiene la lista de dispositivos añadidos al conector biométrico.

Returns:

La lista de dispositivos

---

mostrarDispositivos

```
public void mostrarDispositivos(int tecnologia)
```

Iniciará una búsqueda de dispositivos. Cada vez que se encuentre uno nuevo, enviará un mensaje al Handler del usuario con la acción ConstantesBiometrico.DISPOSITIVO_DESCUBIERTO, en el cual estará el nombre del dispositivo con el identificador ConstantesBiometrico.NOMBRE_DISPOSITIVO_DESCUBIERTO.

---

iniciarMedidas

```
public void iniciarMedidas(String nombreAmigable)
                    throws ErrorBiometricoException
```

Se encargará de conectar a un dispositivo biométrico para comenzar la toma de medidas.

Parameters:

`nombreAmigable` - - Nombre del dispositivo al que se desea conectar para tomar las medidas.

Throws:

`ErrorBiometricoException` - - Si el dispositivo al que se intenta conectar no existe

---

pararMedidas

```
public void pararMedidas(String nombreAmigable)
```

Desconectará un dispositivo y parará la toma de medidas.

Parameters:

`nombreAmigable` - - El dispositivo del cual se va a desconectar

Interface ConexionBiometricoHelper

com.example.pfg.interfazBiometrico

```
public interface ConexionBiometricoHelper
```

Interfaz utilizada para enviar avisos al conector biométrico desde el servicio de medidas que implemente el usuario.

Field Summary		Page
String	**BIOMETRICO_ACTIVADO** Acción para indicar que el dispositivo biométrico está conectado y funcionando correctamente.	128
String	**BIOMETRICO_DESCONEXION_INVOLUNTARIA** Acción para indicar que se ha desconectado el dispositivo biométrico por una causa desconocida.	128
String	**BIOMETRICO_DESCONEXION_VOLUNTARIA** Acción que indica que se ha desconectado el dispositivo biométrico por petición del usuario.	128
String	**NUEVA_MEDIDA_STRING** Acción que indica que hay se ha recibido una nueva medida.	127
String	**TIPO_MEDIDA** Constante para identificar un tipo de medida en el bundle de datos.	128
String	**VALOR_MEDIDA** Constante para identificar el valor de una medida en el bundle de datos.	128

Method Summary		Page
void	**avisarABiometrico**(String evento) Se utiliza para avisar al interfaz biométrico de alguna acción que haya sucedido.	128
void	**avisarABiometrico**(String evento, Bundle datos) Se utiliza para avisar al interfaz biométrico de alguna acción que haya sucedido y además se necesiten incorporar datos adicionales a esa acción.	129

Field Detail

NUEVA_MEDIDA_STRING

```
public static final String NUEVA_MEDIDA_STRING
```

Acción que indica que hay se ha recibido una nueva medida.

BIOMETRICO_ACTIVADO

```
public static final String BIOMETRICO_ACTIVADO
```

Acción para indicar que el dispositivo biométrico está conectado y funcionando correctamente.

BIOMETRICO_DESCONEXION_INVOLUNTARIA

```
public static final String BIOMETRICO_DESCONEXION_INVOLUNTARIA
```

Acción para indicar que se ha desconectado el dispositivo biométrico por una causa desconocida.

BIOMETRICO_DESCONEXION_VOLUNTARIA

```
public static final String BIOMETRICO_DESCONEXION_VOLUNTARIA
```

Acción que indica que se ha desconectado el dispositivo biométrico por petición del usuario.

TIPO_MEDIDA

```
public static final String TIPO_MEDIDA
```

Constante para identificar un tipo de medida en el bundle de datos.

VALOR_MEDIDA

```
public static final String VALOR_MEDIDA
```

Constante para identificar el valor de una medida en el bundle de datos.

Method Detail

avisarABiometrico

```
void avisarABiometrico(String evento)
```

Se utiliza para avisar al interfaz biométrico de alguna acción que haya sucedido. Para ello se deben utilizar las constantes que pone a disposición esta interfaz. La forma de comunicarse con el interfaz biométrico es haciendo uso de un LocalBroadcastManager.

Parameters:

`evento` - - Constante con el tipo de evento sucedido

avisarABiometrico

```
void avisarABiometrico(String evento,
                       Bundle datos)
```

Se utiliza para avisar al interfaz biométrico de alguna acción que haya sucedido y además se necesiten incorporar datos adicionales a esa acción. Para ello se deben utilizar las constantes que pone a disposición esta interfaz. La forma de comunicarse con el interfaz biométrico es haciendo uso de un LocalBroadcastManager.

Parameters:

`evento` - - El evento ocurrido

`datos` - - Los datos adicionales

Class ConstantesBiometrico

com.example.pfg.interfazBiometrico

```
java.lang.Object
  └ com.example.pfg.interfazBiometrico.ConstantesBiometrico
```

```
public class ConstantesBiometrico
extends Object
```

Define constantes referentes a todo lo relacionado con la conexión biométrica.

Field Summary		Page
static int	BUSCANDO_DISPOSITIVOS	131
static int	BUSQUEDA_FINALIZADA	131
static int	DISPOSITIVO_ACTIVADO	131
static int	DISPOSITIVO_DESCONECTADO_INVOLUNTARIO	131
static int	DISPOSITIVO_DESCONECTADO_VOLUNTARIO	131
static int	DISPOSITIVO_DESCUBIERTO	131
static String	DISPOSITIVO_GENERADOR_EVENTO	131
static String	NOMBRE_DISPOSITIVO_DESCUBIERTO	131
static String	NUEVA_MEDIDA_STRING	130
static String	PARAMETROS_SERVICIO	131
static int	SOLICITAR_CONEXION_BT	131

Constructor Summary	Page
ConstantesBiometrico()	132

Field Detail

NUEVA_MEDIDA_STRING

```
public static final String NUEVA_MEDIDA_STRING
```

SOLICITAR_CONEXION_BT

```
public static final int SOLICITAR_CONEXION_BT
```

BUSCANDO_DISPOSITIVOS

```
public static final int BUSCANDO_DISPOSITIVOS
```

DISPOSITIVO_DESCUBIERTO

```
public static final int DISPOSITIVO_DESCUBIERTO
```

NOMBRE_DISPOSITIVO_DESCUBIERTO

```
public static final String NOMBRE_DISPOSITIVO_DESCUBIERTO
```

BUSQUEDA_FINALIZADA

```
public static final int BUSQUEDA_FINALIZADA
```

DISPOSITIVO_ACTIVADO

```
public static final int DISPOSITIVO_ACTIVADO
```

DISPOSITIVO_DESCONECTADO_INVOLUNTARIO

```
public static final int DISPOSITIVO_DESCONECTADO_INVOLUNTARIO
```

DISPOSITIVO_DESCONECTADO_VOLUNTARIO

```
public static final int DISPOSITIVO_DESCONECTADO_VOLUNTARIO
```

DISPOSITIVO_GENERADOR_EVENTO

```
public static final String DISPOSITIVO_GENERADOR_EVENTO
```

PARAMETROS_SERVICIO

```
public static final String PARAMETROS_SERVICIO
```

Constructor Detail

ConstantesBiometrico

```
public ConstantesBiometrico()
```

Class Dispositivo

com.example.pfg.interfazBiometrico

java.lang.Object

└ **com.example.pfg.interfazBiometrico.Dispositivo**

---

public class **Dispositivo**

extends Object

Modelo de dispositivo del que hace uso el paquete biométrico.

Field Summary		Page
static int	TECNOLOGIA_BLUETOOTH	134

Constructor Summary	Page
Dispositivo(String nombre, int tecnologia, Handler handlerDispositivo) Crea un nuevo dispositivo con sus parámetros correspondientes configurados.	134

Method Summary		Page
void	arrancarTimerMedidas() Arranca el timer interno de este dispositivo que determinará si se puede enviar una medida o no dependiendo si ha pasado el periodo de tiempo correspondiente entre una medida y otra.	137
String	getAddressDispositivo()	137
Handler	getHandlerDispositivo()	135
String	getNombreAmigable()	135
String	getNombreReal()	136
ParametrosServicio	getParametrosServicioMedidas()	137
int	getPeriodoMedidas()	136
boolean	getSePuedeEnviarMedida()	137
Service	getServicioMedidas()	137
int	getTecnologia()	136
boolean	isConnected()	135
void	pararTimerMedidas() Para el timer de medidas	137
void	setAddressDispositivo(String addressDispositivo) Establece la dirección física de este dispositivo	137

void	**setConnected**(boolean isConnected) Cambia el estado de conexión del dispositivo	*135*
void	**setHandlerDispositivo**(Handler handlerDispositivo) Asocia a este dispositivo un handler por el cual se mandarán mensajes relacionados con este dispositivo.	*135*
void	**setNombreAmigable**(String nombreAmigable) Da un nombre al dispositivo fácilmente reconocible por el usuario	*135*
void	**setNombreReal**(String nombreReal) Da el nombre de fábrica que trae el dispositivo, o aquel por el cual se puede reconocer en un determinado entorno (por ejemplo, al buscar dispositivos por determinada tecnología)	*136*
void	**setParametrosServicioMedidas**(ParametrosServicio parametrosServicioMedidas)	*138*
void	**setPeriodoMedidas**(int periodo) Establece el periodo de medidas del dispositivo biométrico	*136*
void	**setSePuedeEnviarMedida**(boolean enviarMedida) Establece si este dispositivo está preparado para enviar una determinada medida	*136*
void	**setServicioMedidas**(Service servicioMedidas)	*137*
void	**setTecnologia**(int tecnologia) Establece qué tecnología utiliza este dispositivo Para ello, deben usarse las constantes que proporciona esta clase	*136*

Field Detail

TECNOLOGIA_BLUETOOTH

```
public static final int TECNOLOGIA_BLUETOOTH
```

Constructor Detail

Dispositivo

```
public Dispositivo(String nombre,
                   int tecnologia,
                   Handler handlerDispositivo)
```

Crea un nuevo dispositivo con sus parámetros correspondientes configurados.

Parameters:

`nombre` - - Nombre del dispositivo

`tecnologia` - - Tecnología que utiliza el dispositivo

`handlerDispositivo` - - Handler para manejar eventos relacionados con este dispositivo

Method Detail

setConnected

`public void **setConnected**(boolean isConnected)`

Cambia el estado de conexión del dispositivo

Parameters:

`isConnected` - - boolean indicando si está conectado o no

---

isConnected

`public boolean **isConnected**()`

---

setHandlerDispositivo

`public void **setHandlerDispositivo**(Handler handlerDispositivo)`

Asocia a este dispositivo un handler por el cual se mandarán mensajes relacionados con este dispositivo.

Parameters:

`handlerDispositivo` - - la referencia al handler

---

getHandlerDispositivo

`public Handler **getHandlerDispositivo**()`

---

setNombreAmigable

`public void **setNombreAmigable**(String nombreAmigable)`

Da un nombre al dispositivo fácilmente reconocible por el usuario

---

getNombreAmigable

`public String **getNombreAmigable**()`

---

setNombreReal

```
public void setNombreReal(String nombreReal)
```

Da el nombre de fábrica que trae el dispositivo, o aquel por el cual se puede reconocer en un determinado entorno (por ejemplo, al buscar dispositivos por determinada tecnología)

---

getNombreReal

```
public String getNombreReal()
```

---

setTecnologia

```
public void setTecnologia(int tecnologia)
```

Establece qué tecnología utiliza este dispositivo Para ello, deben usarse las constantes que proporciona esta clase

---

getTecnologia

```
public int getTecnologia()
```

---

setPeriodoMedidas

```
public void setPeriodoMedidas(int periodo)
```

Establece el periodo de medidas del dispositivo biométrico

Parameters:

`periodo` - - El periodo en milisegundos

---

getPeriodoMedidas

```
public int getPeriodoMedidas()
```

---

setSePuedeEnviarMedida

```
public void setSePuedeEnviarMedida(boolean enviarMedida)
```

Establece si este dispositivo está preparado para enviar una determinada medida

---

getSePuedeEnviarMedida

```
public boolean getSePuedeEnviarMedida()
```

---

setAddressDispositivo

```
public void setAddressDispositivo(String addressDispositivo)
```

Establece la dirección física de este dispositivo

---

getAddressDispositivo

```
public String getAddressDispositivo()
```

---

arrancarTimerMedidas

```
public void arrancarTimerMedidas()
```

Arranca el timer interno de este dispositivo que determinará si se puede enviar una medida o no dependiendo si ha pasado el periodo de tiempo correspondiente entre una medida y otra. Hará uso del periodo establecido

---

pararTimerMedidas

```
public void pararTimerMedidas()
```

Para el timer de medidas

---

getServicioMedidas

```
public Service getServicioMedidas()
```

---

setServicioMedidas

```
public void setServicioMedidas(Service servicioMedidas)
```

---

getParametrosServicioMedidas

```
public ParametrosServicio getParametrosServicioMedidas()
```

---

setParametrosServicioMedidas

```
public void setParametrosServicioMedidas(ParametrosServicio parametrosServicio
Medidas)
```

Class ErrorBiometricoException

com.example.pfg.interfazBiometrico

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ com.example.pfg.interfazBiometrico.ErrorBiometricoException
```

All Implemented Interfaces:

Serializable

```
public class ErrorBiometricoException
extends Exception
```

Constructor Summary	Page
ErrorBiometricoException(String error)	139

Constructor Detail

ErrorBiometricoException

```
public ErrorBiometricoException(String error)
```

Class Medida

com.example.pfg.interfazBiometrico

```
java.lang.Object
  └ com.example.pfg.interfazBiometrico.Medida
```

All Implemented Interfaces:
Serializable

```
public class Medida
extends Object
implements Serializable
```

Modelo de medida utilizado por el paquete biométrico. Las medidas tienen un tipo, un determinado valor, unidad correspondiente y fecha en la que ha sido tomada.

Field Summary		Page
static String	UNIDAD_PULSO	141
static String	UNIDAD_VELOCIDAD	141

Constructor Summary	Page
Medida()	141
Medida(String tipo, String valor, String unidad, long fecha)	141

Method Summary		Page
long	getFecha()	142
String	getTipo()	141
String	getUnidad()	142
String	getValor()	141
void	setFecha(long fecha)	142
void	setTipo(String tipo)	141
void	setUnidad(String unidad) Hacer uso de las constantes proporcionadas por esta clase para establecer la unidad de medida.	142
void	setValor(String valor)	141

Field Detail

UNIDAD_PULSO

```
public static final String UNIDAD_PULSO
```

UNIDAD_VELOCIDAD

```
public static final String UNIDAD_VELOCIDAD
```

Constructor Detail

Medida

```
public Medida(String tipo,
              String valor,
              String unidad,
              long fecha)
```

Medida

```
public Medida()
```

Method Detail

getTipo

```
public String getTipo()
```

---

setTipo

```
public void setTipo(String tipo)
```

---

getValor

```
public String getValor()
```

---

setValor

```
public void setValor(String valor)
```

---

getUnidad

```
public String getUnidad()
```

---

setUnidad

```
public void setUnidad(String unidad)
```

Hacer uso de las constantes proporcionadas por esta clase para establecer la unidad de medida. De no hacerlo el funcionamiento podría no ser adecuado.

---

getFecha

```
public long getFecha()
```

---

setFecha

```
public void setFecha(long fecha)
```

Class ParametrosServicio

com.example.pfg.interfazBiometrico

```
java.lang.Object
  └ com.example.pfg.interfazBiometrico.ParametrosServicio
```

```
public class ParametrosServicio
extends Object
```

La clase ParametrosServicio ofrece la posibilidad de pasar al servicio de medidas creado una serie de parámetros que requiera el usuario porque así lo necesite.

Field Summary		Page
static <any>	CREATOR	143

Constructor Summary	Page
ParametrosServicio()	143

Method Summary		Page
void	addParametro(Object param, String key) Añade un parámetro a la lista.	144
int	describeContents()	144
Object	getParametro(String key) Devuelve un parámetro	144
void	writeToParcel(Parcel arg0, int arg1)	144

Field Detail

CREATOR

```
public static final <any> CREATOR
```

Constructor Detail

ParametrosServicio

```
public ParametrosServicio()
```

Method Detail

addParametro

```
public void addParametro(Object param,
                         String key)
```

Añade un parámetro a la lista.

Parameters:

`param` - - El parámetro.

`key` - - El identificador del parámetros.

---

getParametro

```
public Object getParametro(String key)
```

Devuelve un parámetro

Returns:

Un objeto tipo Object en caso de encontrarlo o null en caso contrario.

---

describeContents

```
public int describeContents()
```

---

writeToParcel

```
public void writeToParcel(Parcel arg0,
                          int arg1)
```

Package com.example.pfg.interfazGrafico

Class Summary		Page
GeneradorGraficas	Clase encargada de generar gráficas con los datos que le pase el usuario.	145
Grafica	Modelo de gráfica utilizada por el paquete gráfico.	148
MedidaGrafica	Modelo de medida utilizado por el paquete gráfico.	150

Class GeneradorGraficas

com.example.pfg.interfazGrafico

```
java.lang.Object
  └ com.example.pfg.interfazGrafico.GeneradorGraficas
```

```
public class GeneradorGraficas
extends Object
```

Clase encargada de generar gráficas con los datos que le pase el usuario.

Constructor Summary	Page
GeneradorGraficas(Context c)	146

Method Summary		Page
void	mostrarGraficas(Grafica graficaLinea, Grafica graficaBarra, LinearLayout layoutGrafica) Representa las gráficas dadas en el layout dado.	146

Constructor Detail

GeneradorGraficas

```
public GeneradorGraficas(Context c)
```

Method Detail

mostrarGraficas

```
public void mostrarGraficas(Grafica graficaLinea,
                            Grafica graficaBarra,
                            LinearLayout layoutGrafica)
```

Representa las gráficas dadas en el layout dado.

Parameters:

`graficaLinea` - - Gráfica que se representará en forma de línea.

`graficaBarra` - - Gráfica que se representará en forma de barra. Puede pasarse como null para no representar nada.

`layoutGrafica` - - El layout en el cual se va a realizar la representación gráfica.

Class Grafica

com.example.pfg.interfazGrafico

```
java.lang.Object
  └ com.example.pfg.interfazGrafico.Grafica
```

All Implemented Interfaces:
Serializable

```
public class Grafica
extends Object
implements Serializable
```

Modelo de gráfica utilizada por el paquete gráfico.

Constructor Summary	Page
Grafica(List<MedidaGrafica> medidas, String nombreGrafica) Crea una instancia de gráfica.	148

Method Summary		Page
Date[]	**getFechasEjeX**() Devuelve las fechas en las que se toman las medidas.	149
String[]	**getLocalizaciones**() Devuelve las localizaciones existentes en la lista de medidas.	149
String	**getNombreGrafica**()	149
int	**getNumValores**() Devuelve el número de medidas totales a representar.	149
Double[]	**getValoresEjeY**() Devuelve los valores de las medidas.	149
void	**setNombreGrafica**(String nombreGrafica) Da un nombre a la gráfica.	149

Constructor Detail

Grafica

```
public Grafica(List<MedidaGrafica> medidas,
               String nombreGrafica)
```

Crea una instancia de gráfica.

Parameters:

`medidas` - - La lista de medidas que se van a representar.

`nombreGrafica` - - El nombre de esta gráfica.

Method Detail

setNombreGrafica

```
public void setNombreGrafica(String nombreGrafica)
```

Da un nombre a la gráfica.

---

getNombreGrafica

```
public String getNombreGrafica()
```

---

getLocalizaciones

```
public String[] getLocalizaciones()
```

Devuelve las localizaciones existentes en la lista de medidas.

---

getValoresEjeY

```
public Double[] getValoresEjeY()
```

Devuelve los valores de las medidas.

---

getFechasEjeX

```
public Date[] getFechasEjeX()
```

Devuelve las fechas en las que se toman las medidas.

---

getNumValores

```
public int getNumValores()
```

Devuelve el número de medidas totales a representar.

Class MedidaGrafica

com.example.pfg.interfazGrafico

```
java.lang.Object
  └ com.example.pfg.interfazGrafico.MedidaGrafica
```

All Implemented Interfaces:
Serializable

```
public class MedidaGrafica
extends Object
implements Serializable
```

Modelo de medida utilizado por el paquete gráfico.

Constructor Summary	Page
MedidaGrafica(String tipo, String valor, String localizacion, long fecha)	150

Method Summary		Page
long	getFecha()	151
String	getLocalizacion()	151
String	getTipo()	151
String	getUnidad()	151
String	getValor()	151
void	setFecha(long fecha)	151
void	setLocalizacion(String localizacion)	152
void	setTipo(String tipo)	151
void	setUnidad(String unidad)	151
void	setValor(String valor)	151

Constructor Detail

MedidaGrafica

```
public MedidaGrafica(String tipo,
                     String valor,
                     String localizacion,
                     long fecha)
```

Method Detail

getTipo

```
public String getTipo()
```

---

setTipo

```
public void setTipo(String tipo)
```

---

getValor

```
public String getValor()
```

---

setValor

```
public void setValor(String valor)
```

---

getUnidad

```
public String getUnidad()
```

---

setUnidad

```
public void setUnidad(String unidad)
```

---

getFecha

```
public long getFecha()
```

---

setFecha

```
public void setFecha(long fecha)
```

---

getLocalizacion

```
public String getLocalizacion()
```

---

setLocalizacion

```
public void setLocalizacion(String localizacion)
```
